TAMPERE UNIVERSITY OF TECHNOLOGY

JANNE HELKALA

VARIABLE LENGTH INSTRUCTION COMPRESSION ON TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Examiners: Prof. Jarmo Takala and D.Sc. Pekka Jääskeläinen
Examiners and topic approved in the Faculty of Computing and Electrical Engineering Council meeting
4th of December 2013

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Signal Processing and Communications Engineering
JANNE HELKALA : Variable Length Instruction Compression on Transport Triggered Architectures
Master of Science Thesis, 58 pages
June 2014
Major: Programmable platforms and devices
Examiners: Prof. Jarmo Takala and D.Sc. Pekka Jääskeläinen
Keywords: processors, parallel processing, hardware description

The *Static Random-Access Memory* (SRAM) modules used for embedded microprocessor devices consume a large portion of the whole system's power. The memory module consumes static power on keeping awake and dynamic power on memory accesses. The power dissipation of the instruction memory can be limited by using code compression methods, which reduce the memory size. The compression may require the use of variable length instruction formats in the processor. The power-efficient design of variable length instruction fetch and decode units is challenging for static multiple-issue processors, because such architectures have simple hardware to begin with, as they aim for very low power consumption on embedded platforms. The power saved by using these compression approaches, which necessitate more complex logic, is easily lost on inefficient processor design.

This thesis proposes an implementation for instruction template-based compression, its decompression and two instruction fetch design alternatives for variable length instruction encoding on *Transport Triggered Architecture* (TTA), a static multiple-issue exposed data path architecture. Both of the new fetch and decode units are integrated into the *TTA-based Co-design Environment* (TCE), which is a toolset for rapid designing and prototyping of processors based on TTA.

The hardware description of the fetch units is verified on a register transfer level and benchmarked using the CHStone test suite. Furthermore, the fetch units are synthesized on a 40 nm standard cell *Application Specific Integrated Circuit* (ASIC) technology library for area, performance and power consumption measurements. The power cost of the variable length instruction support is compared to the power savings from memory reduction, which is evaluated using HP Labs' CACTI tool.

The compression approach reaches an average program size reduction of 44% at best with a set of test programs, and the total power consumption of the system is reduced. The thesis shows that the proposed variable length fetch designs are sufficiently low-power oriented for TTA processors to benefit from the code compression.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Signaalinkäsittelyn ja tietoliikennetekniikan koulutusohjelma
JANNE HELKALA: Siirtoliipaisuarkkitehtuurin muuttuvanmittaisten käskyjen pakkaus
Diplomityö, 58 sivua
Kesäkuu 2014
Pääaine: Ohjelmoitavat alustat ja laitteet
Tarkastajat: prof. Jarmo Takala ja TkT Pekka Jääskeläinen
Avainsanat: prosessorit, rinnakkaiskäsittely, laitteistokuvaus

Sulautettujen mikroprosessorien käyttämät staattiset muistit kuluttavat suuren osan systeemin tehosta johtuen staattisesta tehonkulutuksesta muistia ylläpitäessä sekä dynaamisesta tehonkulutuksesta muistista lukiessa. Käskymuistin tehonkulutusta voi rajoittaa käyttämällä ohjelmakoodin pakkausta, joka pienentää muistin kokoa. Koodinpakkaus voi tarvita muuttuvanmittaisia käskyjä prosessorissa. Muuttuvanmittaisia käskyjä käyttävän prosessorin käskynhaku- ja dekoodausyksiköiden suunnittelu on haastavaa tehonkulutuksen kannalta staattisille prosessoriarkkitehtuureille, sillä sellaisten arkkitehtuurien laitteistokuvaus on alkujaan yksinkertainen. Yksinkertaisella arkkitehtuurilla tähdätään mahdollisimman pieneen tehonkulutukseen sulautetuissa järjestelmissä. Koodinpakkauksella säästetty teho menee helposti hukkaan huonosti suunnitellussa laitteistokuvauksessa.

Tämä työ ehdottaa toteutusta käskymalleihin perustuvalle koodinpakkaukselle, sen dekompressorille ja kahdelle muuttuvanmittaisia käskyjä tukevalle käskynhakuyksikölle. Toteutukset tehdään siirtoliipaisuarkkitehtuurille *(Transport Triggered Architecture, TTA)*, joka on staattinen, helposti räätälöitävä arkkitehtuuri jossa ohjelmoijalla on näkyvyys prosessorin datapolulle. Uusien käskynhakuyksiköiden ja dekoodausyksikön toteutukset integroidaan *TTA-based Co-design Environment* (TCE) -työkaluympäristöön, jota käytetään TTA-prosessorien nopeaan suunnitteluun ja prototyyppauksen.

Käskynhakuyksiköiden laitteistokuvaus on varmennettu rekisterisiirtotasolla ja testattu käyttäen CHStone-testipenkkiä. Sen lisäksi käskynhakuyksiköt on syntesoitu 40 nm standardisoluteknologiakirjastoja käyttäen pinta-ala-, nopeus- ja tehonkulutusestimaattien saamiseksi. Muuttuvanmittaisten käskyjen vaatiman logiikan tehonkulutusta verrataan muistin pienennyksestä saatavaan tehonsäästöön, joka arvioidaan HP Labsin CACTI-työkalulla.

Koodinpakkausmenetelmä saavuttaa parhaimmillaan 44%:n keskimääräisen ohjelmakoodin pakkauksen testiohjelmilla, ja sulautetun prosessorin kokonaistehonkulutus laskee. Työ osoittaa, että käskynhakuyksiköt ovat tarpeeksi pienitehoisia TTA-prosessoreille, jotta koodinpakkauksesta hyödytään.

PREFACE

The work in this M.Sc. thesis was carried out at the Department of Pervasive Computing at Tampere University of Technology as a part of the Parallel Acceleration Project (ParallaX).

First, I would like to thank my examiners: Prof. Jarmo Takala for letting me work on this project and D.Sc. Pekka Jääskeläinen for his valuable pieces of advice on how to write scientific text. Furthermore, thanks go to my colleagues in the Customized Parallel Computing group at TUT for giving motivation and sharing insightful ideas. Especially Timo Viitanen contributed a lot during the design phase of the hardware units, which deserves a mention. I am also grateful to Tommi Zetterman from Nokia Research Center for his help on the power measurements in this thesis. Last but not least, thanks to my family for giving me encouragement throughout the work.

Tampere, May 5, 2014

JANNE HELKALA

CONTENTS

SYMBOLS AND ABBREVIATIONS

a_w	Alignment bit width
f_{clk}	Clock frequency
Ci_w	Current instruction width
E_{dyn}	Dynamic energy per read port
P_{dyn}	Dynamic read power per read port
$Imem_w$	Instruction memory width
ITF_w	Instruction template bit field width
I_{max}	Maximum instruction size
q	Quantum, minimum instruction size
Rb_w	Ring buffer width
ADF	Architecture Definition File
ALU	Arithmetic Logic Unit
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
CISC	Complex Instruction-Set Computers
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
CU	Control Unit
EPIC	Explicitly Parallel Instruction Computing
FPGA	Field-Programmable Gate Array
FU	Function Unit
GPP	General Purpose Processor
GPU	Graphical Processing Unit
HAT	Heads-And-Tails
HDB	Hardware Data Base
HDL	Hardware Description Language
HP	High Performance
ID	Identification
IDF	Implementation Definition File

ILP	Instruction-Level Parallelism
ISA	Instruction-Set Architecture
ITF	Instruction Template Field
LIMM	Long IMMediate
LLVM	Low Level Virtual Machine
LSB	Least Significant Bit
LSTP	Low STandby Power
LSU	Load-Store Unit
LUT	Look-Up Table
MIPS	Microprocessor without Interlocked Pipeline Stages
MSB	Most Significant Bit
NOP	No-OPeration
PC	Program Counter
PIG	Program Image Generator
ProDe	Processor Designer
ProGe	Processor Generator
RA	Return Address
RAW	Read After Write
RISC	Reduced Instruction-Set Computers
RP	Read Pointer
SA	Shift Amount
SRAM	Static Random-Access Memory
TCE	TTA-based Co-design Environment
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuits
VLIW	Very Long Instruction Word
TPEF	TTA Program Exchange Format
TTA	Transport Triggered Architectures
TVLIW	Tagged VLIW
WAR	Write After Read
WAW	Write After Write
WP	Write Pointer

1. INTRODUCTION

Modern systems-on-a-chip are becoming more and more advanced as an increasing amount of CMOS transistors can be fit on a single integrated circuit. Larger programs can be stored on the on-chip memories of devices, which consume a significant portion of the system's power. This makes it important to focus on reducing the memory accesses and memory size to reach a better power consumption level on the whole, especially on embedded, battery-powered devices which aim for very low power consumption levels.

The power consumption of a circuit is divided into two categories: dynamic power and static power. The majority of the power dissipated in an integrated circuit is due to dynamic activity: net switching power, internal cell power and short-circuit power during logic transitions in the transistors [1]. However, the proportion of static power dissipation, also known as leakage power, is quickly growing towards half of all power used as the deep submicron technology nodes continue to decrease in size [2]. The smaller the technology, the more leakage will be present.

The program code, which is stored on SRAM for embedded microprocessors, is an important aspect to consider for power savings. If *high performance* (HP) SRAM is used on the chip, a substantial amount of current leakage is present [3]. Slower *low standby power* (LSTP) SRAM can be used to avoid large leakage, but LSTP memory cells have higher on-currents, consuming more dynamic power as a trade-off. For either technology used, reducing the size of the memory via program code compression is beneficial: HP SRAM leaks less current when the memory module is smaller, while less dynamic power is used on expensive LSTP memory read-accesses if multiple instructions can be read per clock cycle.

Static multiple-issue architectures such as TTA [4] can gain a lot of power savings from program code compression due to their long instruction formats, which require large on-chip memories to store the program code. The challenge brought by some code compression approaches, such as instruction template-based compression, is the need for variable length instruction fetch and decode units. They are difficult to design power-efficiently on embedded devices employing static-scheduled processors, which have fairly simple fetch and decode hardware as the starting point. If low-power variable length support hardware can be designed for the processor, power can be saved with sufficient memory reduction.

In this master's thesis project, an instruction template-based compression method used for *no-operation* (NOP) removal was designed for use in TTA processors. The required power-efficient variable length instruction encoding fetch and decompression units were implemented and documented. Furthermore, the ability to use these new designs and the compression method were integrated to *TTA-based Co-design Environment* (TCE), which is a toolset for rapid creation of customized TTA processors.

Two alternative fetch unit designs are synthesized and benchmarked on a 40 nm standard cell ASIC technology for area, performance and power consumption measurements. The efficiency of the code compression is gauged by creating a custom TTA processor with a maximum instruction length of 256 bits and minimum length of 32 bits for the CHStone [5] test suite. Each test program's code for the processor is compressed using four and eight instruction templates. The feasibility of the implementation is evaluated by comparing the power consumption of each test's program memory pre- and post-compression with HP Labs' CACTI [6, 7] and comparing the savings with the instruction fetch units' power consumption.

A conference paper [8] was written based on the work in this thesis for the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS) XIV. The paper was targeted for the applications, systems, architectures and processors track. It was accepted with positive reviews.

The structure of this thesis is as follows: Chapter 2 is a preamble with computer architecture concepts and terminology relevant to this thesis. Chapter 3 discusses variable length instruction encoding, instruction compression and the importance of NOP removal. Chapter 4 is a related work chapter which reviews existing variable length instruction encoding styles and instruction compression mechanisms. Chapter 5 details how instruction template-based compression is used in this thesis and specifies two proposed alternative fetch designs and the decompression routine. Chapter 6 explains how the new hardware was integrated to TCE, so they can be easily adopted to processor designs. Chapter 7 tells about the verification process and lists benchmark results from the implemented compression method and fetch units. The future work is contemplated in Chapter 8 and the thesis is concluded in Chapter 9.

2. MULTIPLE-ISSUE ARCHITECTURES

The performance of processors has been successfully on the rise for the past several decades thanks to new techniques enabled by the increase in chip density. As the number of available transistors on processor chips has increased, it has been possible to incorporate features such as pipelining, increased word and instruction sizes, inclusion of caches and multiple computation resources on a single processor [9]. The lattermost is perhaps the most important, as it has enabled a large amount of *Instruction-Level Parallelism* (ILP) in processors. ILP means that a processor is able to execute several lowest level machine operations in parallel, whether it is by pipelining or by using multiple processing resources simultaneously. These machine operations include, for example, memory load and stores, *Arithmetic Logic Unit* (ALU) operations, register reads and writes and floating point multiplications.

What kind of processing resources are available and how they are used in a processor are defined by its *computer organization*, also called the computer's *microarchitecture*. The microarchitecture defines how a processor implements a given *Instruction Set Architecture* (ISA), the machine language for the processor. An example of the ISA is the x86 instruction set [10], and examples of processors with different internal designs, that is, different microarchitectures, are Intel Pentium and AMD Athlon, which both implement nearly identical versions of the x86 ISA. Furthermore, the broad term *architecture* is used to refer to all of the following aspects of the computer system: the microarchitecture, the ISA and the rest of the processor's hardware components used in the system, such as data paths, memory controllers, memory hierarchies and units for software features. In other words, the architecture specifies all the parts and their relations in a computer system.

A processor is said to be multiple-issue if a single instruction stream feeds several processing resources at a time. For example, an n-issue processor would issue n operations per cycle to the processor's hardware resources. Multiple-issue processors come mainly in two styles: dynamic and static [11]. Dynamic and static multiple-issue architectures search for ILP in two very different ways. Whether a processor architecture is dynamic or static is determined by how the instructions are scheduled for execution. The schedule of a program defines what is the sequence of instructions that is given to the *Central Processing Unit* (CPU) to implement the program. In dynamic architectures the CPU is given a sequential program and its

task is to execute the program correctly while searching for ILP during run-time. That is, as the program is running, the hardware needs to understand the data and control dependencies between instructions so it can assign multiples of them simultaneously to available processing resources, while guaranteeing correct execution. A term often used for processors, which use dynamic scheduling techniques to execute multiple instructions simultaneously on different processing resources, is *superscalar processors*. The majority of modern, general purpose CPUs, for example, ARM Cortex-A9, Intel Core i7 and IBM Power7, are superscalar processors [12].

The design philosophy for dynamically scheduling processors is to find ILP with hardware, whereas in static architectures ILP is found by software, by the compiler. In a static architecture, when the program code arrives to the CPU, it is already in a form which has ILP exposed when executed. The data and control dependencies between instructions are found in scheduling during compile-time. As this process is done during compile-time, it allows the scheduler to inspect the program at different levels of granularity to arrange the code for ILP, such as basic block level, region level or the whole program level. ILP is extracted by using methods such as trace scheduling, loop unrolling, profiling and software pipelining. By contrast, in dynamically scheduled architectures, the ILP is hidden in the program code which arrives to the CPU. Some compile-time scheduling is done by the compiler even in superscalar architectures to make it easier for the hardware to rediscover the ILP hidden in the program code, as the program control hardware has a much narrower scheduling window during run-time to extract the ILP [13]. From this it can be gathered, that the bipartition to static and dynamic architectures is not unambiguous, as these design styles sometimes borrow improvements from each other to their own advantage.

The majority of the engineering effort in these two design philosophies is quite distinctly divided into hardware and software. The control hardware is difficult to implement for dynamic scheduling as it has to resolve different kinds of dependencies for instructions to avoid hazards. This becomes even more complex in multi-core architectures and in architectures, which support out-of-order scheduling of instructions. On the other hand, in static architectures, there are often more operations per instruction bundle and more processing resources to divide operations to. The scheduler needs to select the correct operations for processing resources for each instruction, that is, do instruction bundling or instruction scheduling. However, the actual engineering complexity in a static architecture lies in generating program code that exploits ILP well, not just in picking the correct operations for instruction bundles [13]. Because of sophisticated scheduling algorithms and a larger instruction window to schedule instructions from, static architectures such as *Very Long Instruction Word* (VLIW) have been said to have potential for better ILP in than

superscalar architectures [14].

Superscalar processors are often chosen for applications which need a significant amount of control code execution, whereas static architecture is used when there is a lot of program code which parallelizes well, such as in extensive calculation tasks. Another notable distinction between superscalar and static architectures is the size: superscalar CPUs are complex due to the hardware which searches for ILP dynamically, while static microprocessors are often simple and power-efficient. However, as more and more transistors have become available with better density, static architectures have started becoming increasingly diverse as well. More processing resources can be added, there can be more buses and instructions are longer. It is not strange to see specialized extra hardware for instruction compression or other benefits in static-scheduled embedded-device microprocessors. From the increase of transistor budget and advances in compilation techniques, the aforementioned architectural design philosophy called VLIW emerged in the early 1980s.

2.1 Very Long Instruction Word

VLIW processors were first introduced in 1980s by Fisher in [15], but there wasn't enough transistor density until the year 1990 for VLIW on a silicon chip [13]. Adaptations of VLIW appear in commercial platforms such as Analog Devices' *Digital Signal Processor* (DSP) TigerSHARC [16], Qualcomm's Hexagon DSP-based CPU architecture [17], Transmeta Corporation's x86-compatible Efficeon mobile platform processor [18] and Intel's enterprise server processors with their Itanium series [12]. Intel has named their evolution of the VLIW design philosophy as *Explicitly Parallel Instruction Computing* (EPIC) [19], which includes architectural concepts to increase ILP and provides object code compatibility between processors. According to Fisher et al. in [13], while many DSP processors are not clean VLIWs, they follow the *Instruction-Set Architecture* (ISA) and design principles closely enough to be a part of the VLIW taxonomy.

While processors can be called to have a VLIW architecture, it also is a design philosophy which states to expose instruction-level parallelism in the architecture to the programmer. This parallelism is achieved with pipelining, an advanced compiler, multiple processing resources called *Function Units* (FU) that execute instructions in parallel and simple encodings, which are a selection of instructions for the processor that avoid idiosyncrasy. As per the name *VLIW*, the instructions can be very long and may contain dozens of operations per instruction for FUs. [13]

As VLIW is a static architecture, it avoids the out-of-order hardware scheduler, is easier to design and has good scalability. Existing designs can be mutated easily into new ones because of the simpler hardware. Often VLIW-style designs also have lower power consumption, shorter design cycles and require less hardware. While the

hardware side is simple, the trade-off is that it requires a good compiler to exploit the ILP. VLIWs are also known to have difficulty with backwards compatibility and upgradeability, as the compiler has to create target-specific code to enable high performance. That is, the program code is compiled into a binary that works in one specific architecture only, and cannot be used in other VLIW processors unless they are very similar in issue amounts, registers, bus widths, latency and FU availability. Practically a recompilation is always required if the architecture changes.

While VLIW can scale up to large instruction lengths, its scalability is limited by the data path connectivity with the processing resources and the register file complexity. Bypassing values from all the inputs of FUs to their outputs requires a full crossbar connectivity. The complexity of this connection network grows quadratically as FUs are added. Because of this large network, there is often much more extra bandwidth available than is actually used. In addition, each of the FUs require two read ports and one write port on the register file, resulting in a lot of connectivity for a worst case situation. In order to simplify the connectivity structure of the VLIW architecture, Corporaal et al. proposed to use an architecture called the *Transport Triggered Architecture* (TTA). [20]

2.2 Transport Triggered Architecture

A processor architecture named *Conditional MOVE* which uses transport programming was first proposed by Lipovski in 1970s [21]. In the 1990s Corporaal et al. researched the TTA architecture in Delft University of Technology in Netherlands in a project called MOVE, producing an example processor implementation called MOVE32INT [22]. One of the key aspects that MOVE project explored was reducing VLIW's register file connection complexity for TTA. Their results showed that substantial reductions to register file ports could be achieved by using data transport programming instead of operation programming [23]. Further development of the MOVE project's framework was continued at the Tampere University of Technology in Finland until 2002, when the development of *TTA-based Co-design Environment* (TCE) framework began. TCE will be discussed further in Chapter 6.

TTA is a modular processor architecture, which deviates from the well-known *Single Instruction, Multiple Operations* programming paradigm and instead functions as a *Single Instruction, Multiple Transports* machine. In place of operations, instruction bundles contain one or multiple transports or *moves*. As opposed to operations for each FU like in a VLIW architecture, a TTA instruction has a move (operation) for each bus in the architecture for transporting data. An operation is triggered as a side-effect when data is moved to the *triggering port* of a FU. The processor contains an interconnection network, FUs, load-store units, register files, data and program memory, an immediate unit and a *Control Unit* (CU), which

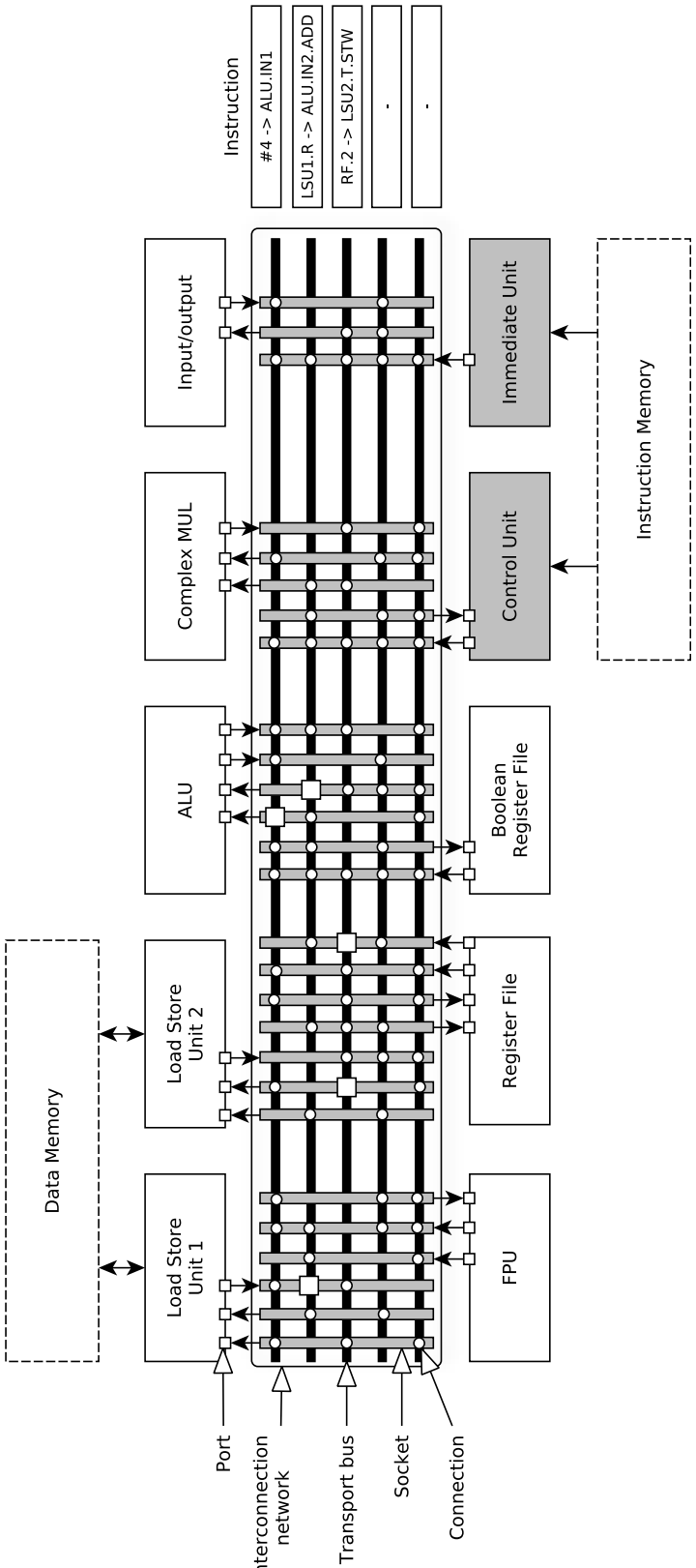

Figure 2.1: Example of a TTA processor's data path. The data path of this processor consists of two load-store units, register files and various function units in addition to the mandatory input/output ports, CU and immediate unit. In the example instruction, three *moves* are programmed on the three first buses and a NOP move occurs on the last two buses. The instruction contains an integer ADD-operation with an immediate and a value loaded from the memory by the load-store unit, and in parallel a move which stores a value from a register file to the memory. The summation in the ALU is triggered by the second move, which is directed to the ALU's *trigger port*. The last two buses are idle. White circles indicate which buses have connections to which function units, and white squares indicate which of those connections are used by the moves in the example instruction.

handles instruction fetching, decompression, decoding and the execution of control flow operations such as *call* and *jump*. The architecture's modularity enables swift generation and easy prototyping of processors for different applications, allowing the designer to add FUs to increase performance and insert buses for more data bandwidth. TTA has been used to create, for example, *Application-Specific Processors* (ASP) [24, 25]. An example of a TTA machine and its data path is depicted in Figure 2.1.

The control unit is of particular interest for this thesis as it contains the fetch unit, an optional decompression unit and the decoder unit. Each of these is a notable unit in a TTA processor's pipeline, which is show in Figure 2.2. The task of the fetch unit is to increment the internal *Program Counter* (PC), handle return address calculation and fetch fixed length instructions from the program memory. These instructions are routed to the decompressor, if compression has been applied to the instructions and requires a decompressor of its own. The decoder generates signals from the instruction bits for the interconnection network, which executes the transports toward hardware units detailed in the instruction bundle.

Just like the VLIW architecture, TTA is a static multiple-issue architecture where the task of exposing the ILP is on the compiler. However the program code is in the form of transports of data between the available processing resources, and as many moves can be issued each cycle as there are buses. TTA's answer to VLIW's scalability issue is in the transport triggered programming paradigm: The hardware control logic that would route operations to correct function units is absent, as the

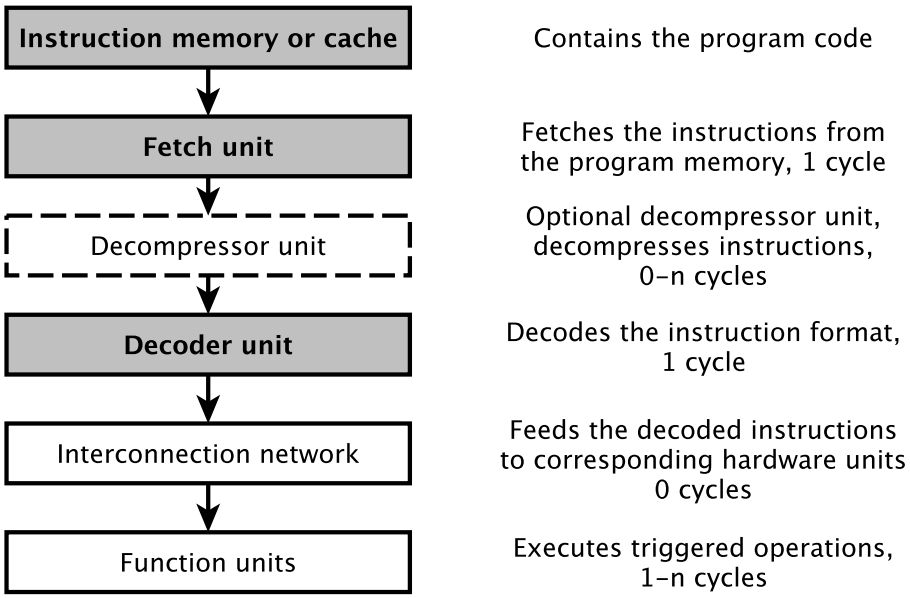

Figure 2.2: A TTA processor's instruction pipeline. The parts modified in this work are indicated in gray.

TTA program directly controls the interconnection network. This model allows the use of two methods of optimization: First, the output of a FU can be directly moved to the input port of a FU through the interconnection network, without storing the value in a register file in between. This optimization is called *software bypassing*. As a by-product, an optimization called *dead result elimination* occurs when all the usages of a value can be bypassed instead of using a register file. In a measurement done in [20], for a certain sufficient VLIW architecture, 50% of register file accesses can be reduced by using bypassing and 35% of results do not need to be written to a register file, thus can be eliminated by dead result elimination. In this context, a sufficient architecture means that the processor did not get significant performance gains from the addition of new FUs. Second, the value of an earlier operation in a FU can be reused by simply storing it in the input port if the FU is not used for anything else in between.

Despite these improvements in TTA, the design model still shares some of VLIW's problems. Programs need to be recompiled if the target architecture changes. The move operations which describe the flow of a program only work with the particular set of FUs, register files and buses in the architecture, which the program was compiled for. Furthermore, while the issue with VLIW's interconnect complexity explosion was alleviated and it enables wider architectures, the instructions reserve a lot of program memory in large machines and concurrently require significant memory bandwidth, which leads to high power consumption.

The main body of a TTA instruction consists of consecutive move slots which contain the transport encoding. In addition, there can be optional fields which are required by *long immediate* (LIMM) support. The instruction format depends on the TTA processor's configuration and the order of the fields and their widths are customizable. An instruction layout, which contains at least one of each field supported by TTA's instruction format for a 2-issue machine, is shown in Figure 2.3.

The instruction is divided into as many move slots as there are buses in the TTA processor. This instruction format example begins with an *Instruction Template Field* (ITF), which is a template selector field telling if this instruction replaces some move slots with LIMM data. The *LIMM destination register field* tells which LIMM register is targeted by this instruction. *Dedicated LIMM fields* can exist to solely contain more LIMM bits, instead of carrying them in move slots.

Each move slot is further divided into three fields: *guard field*, *source field* and *destination field*. The guard field is used if conditional execution of a move is desired. It can contain the numerical *identification* (ID) of a register, the content of which then tells whether to transport the move or not. The source field describes the source of the data transport, such as a FU, but can instead carry short immediate data if the operation requires it. The destination field contains the move's destination unit

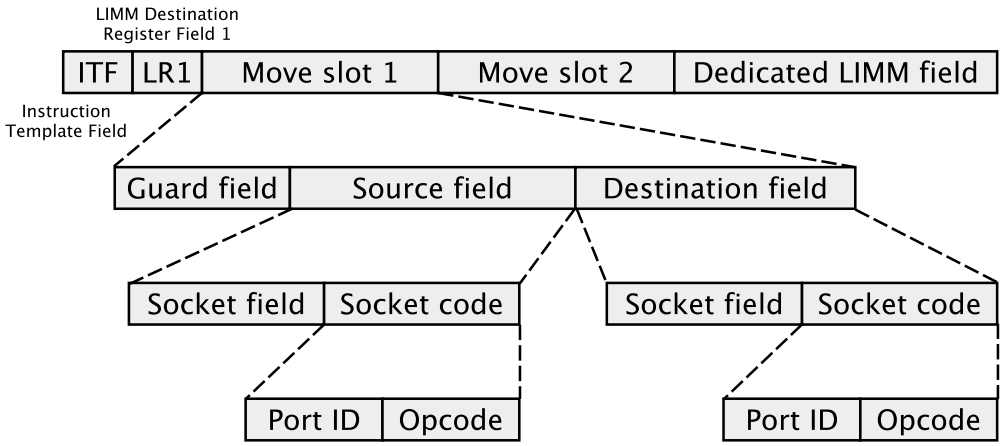

Figure 2.3: An example of a TTA instruction format for a 2-issue TTA processor.

encoding. Both source field and destination field are divided into a *socket field* and a *socket code*, which is further split into a *port ID* and an *opcode*. The socket ID defines the source or destination socket of the move, each FU and other units having their own socket IDs. The port ID is used when multiple ports are connected to a socket, and the opcode defines which register number is used when reading from or writing to a register file. Furthermore, the opcode is used if there are several different operations to choose from in a FU.

2.3 Dependencies and No-Operations

In order to exploit ILP within a program, the dependencies between instructions in the program must be understood. Not all operations or instructions can be arbitrarily chosen to be run in parallel because they may depend on data that other operations require. The dependencies in the program limit the amount of ILP that can be extracted. These dependencies disrupt the smooth program flow in the pipeline and must be resolved either by software or hardware means for correct program execution. Dependencies come in three types: resource dependencies, control dependencies and data dependencies. [12]

Resource dependencies describe a situation where multiple instructions depend on the same hardware resource. For instance, two division operations cannot be executed in parallel on a 2-issue machine if only one FU capable of division exists in the processor architecture. This can limit the amount of parallelism that can be achieved if the division operations need to be executed consecutively or concurrently.

Control dependencies arise due to processor not knowing the result of a branch operation when it is seen in the fetch stage of the processor pipeline. This restricts the processor from sending any more instructions down the pipeline.

Data dependencies are another common type of dependencies and occur due to

dependencies between data in the processor pipeline. Each of the three types of data dependencies are explained briefly below, using registers *R1–5* and instructions *I1–2* that are to be executed consecutively. The name of the data dependency describes in which order the two operations, write and read, are seen in the pipeline and are causing the dependency.

An example of *Read After Write* (RAW) dependency is below, where the multiply instruction *I2* is dependent on the value stored in *R1* which is calculated in the addition instruction *I1*. Due to pipeline latency, *I2* cannot be correctly executed until the new value is in *R1*:

$$I1.\ \mathbf{R1} \leftarrow R2 + R3$$
$$I2.\ R4 \leftarrow \mathbf{R1} \times R5$$

A *Write After Read* (WAR) dependency is a situation where the value in *R1*, which is to be used in the multiply instruction *I1*, might change due to the addition instruction *I2*. If for any reason *I2* were to be executed ahead of *I1*, such as FU latency or code reordering, the multiply instruction would lead to an erroneous result:

$$I1.\ R2 \leftarrow R3 \times \mathbf{R1}$$
$$I2.\ \mathbf{R1} \leftarrow R4 + R5$$

Finally, a *Write After Write* (WAW) dependency describes a situation, where two instructions write their values to the same register *R1*, which may happen in the wrong order due to concurrent execution or FU latency:

$$I1.\ \mathbf{R1} \leftarrow R2 + R3$$
$$I2.\ \mathbf{R1} \leftarrow R4 \times R5$$

The elimination of dependencies happens mostly on software in static architectures, whereas dynamic architectures rely on hardware mechanisms such as out-of-order execution to reorder the instructions correctly on-the-fly. On a static-scheduled architecture, the most cost-efficient method of removing dependencies is code reordering, so parallelism can be found for the program despite dependencies requiring some instructions to be executed in a specific order. [13]

In the worst case, in order to avoid hazards, no-operations may need to be inserted into the pipeline. A NOP is a null operation, which tells the processor or specific FUs to just stay on standby for one cycle. NOPs can be used in each of the dependency type to resolve the dependency between the operations, but it decreases the ILP as useful instructions are not executed in parallel. An example of resolving a resource

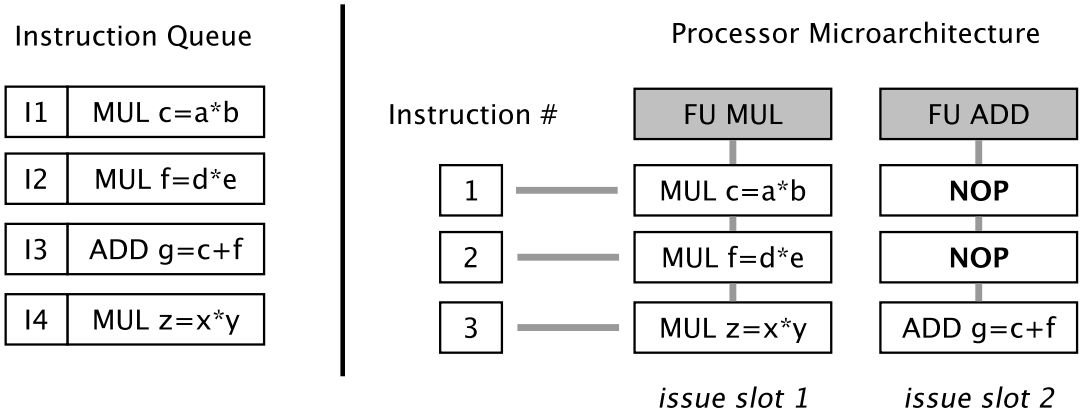

Figure 2.4: A 4-instruction-long program is executed in a 2-issue VLIW machine, which has MUL and ADD FUs. For the sake of an example, each instruction is executed in 1 cycle. The addition instruction $I3$ depends on the values calculated by multiply instructions $I1$ and $I2$. The NOPs are required because of a data dependency, as the ADD-instruction depends on the values c and f, which are being calculated during cycles 1 and 2. Only the ADD-instruction $I3$ and the independent MUL-instruction $I4$ can be executed in parallel because of the dependency.

and a RAW-type data dependency with NOPs is displayed in Figure 2.4.

The amount of NOPs in a program varies depending on how well the given program fits the processor architecture. The NOPs required due to resource dependencies can be limited by customizing the architecture for the applications, so that an optimal amount of FUs is available for the tasks that need to be calculated in the program. For example, a multiplication-intensive program should have a sufficient amount of FUs capable of multiplication. In spite of optimization efforts, large machines which are designed to operate on data in parallel often have a significant amount of NOPs in the program code. This owes to small portions of serial control code, which might not work well in parallel in a large multiple-issue processor architecture dedicated to data manipulation. Furthermore, the more general-purpose design a multiple-issue processor has, the more likely it is to have NOPs from resource dependencies due to suboptimal program-to-machine compatibility.

3. COMPRESSION AND VARIABLE LENGTH INSTRUCTION ENCODING

This chapter explains the idea of program compression and how encoding of instructions is often associated with compression schemes. The principles of supporting variable length instructions on a processor is detailed. Furthermore, the effects of encoding and compression on the hardware are explained.

3.1 Variable Length Instruction Encoding

The main purpose of variable length instruction encoding is to encode some instructions in a smaller amount of bits than others, because instructions with less payload can often be encoded into a smaller representation. For example, an ADD instruction may have several register or immediate operands attached to it which all need to be represented with a set of bits, but a NOP instruction tells the processor to do nothing, which can often be instructed by a smaller amount of bits. This approach is not only useful for NOPs, but other shorter instruction formats requiring less operands can be designed in the ISA as well.

While variable length instruction encoding can shorten instruction sizes, it comes with a cost of increased hardware complexity and necessitates the redesign of the architecture's instruction format. On the architectural level, a design decision is needed on how to allocate the variable length instructions in the memory, which is a part of the *random instruction access support* problem, meaning the need to transfer the program flow to selected instructions in the memory with instructions such as jump and call. The variable sizes of instructions may cause misalignment in the memory, which means that instructions which require jumping to do not begin at addressable memory boundaries. On the hardware level, the fetching of instructions becomes difficult as the beginning and the end of instructions need to be found from the packed memory data in order to dispatch the instructions forward, but also to increment the program counter correctly. The correct timing of instructions such as jump and call becomes complicated due to instruction storage in the fetch unit's buffer. Furthermore, a hardware mechanism for the decompression of the compressed instructions with variable sizes needs to be decided on.

There are numerous ways to implement the microarchitecture to support variable length instruction encoding. Often the instruction's size is tied to the instruction

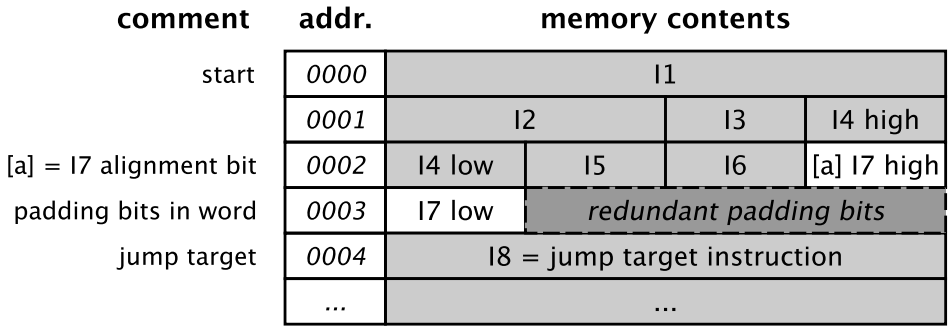

Figure 3.1: An example of a processor's misaligned program memory, containing instructions *I1–I8*. The instruction *I*8 is a jump target which needs to be aligned. Thus *I*7's *alignment bit* = '1', indicating that the bits after it in the memory word are redundant. When *I*7 is dispatched, the fetch unit's logic must realize to increase PC from 0003 to 0004 without interpreting the contents of the padding bits, dispatching *I*8 on the following cycle.

type or a particular template that defines the purpose of the instruction. As for decompression, its particularities depend entirely on the compression method and the processor architecture.

However, random instruction access support is worth detailing to give a better understanding of the processor's instruction memory layout. There is more than one approach how to refer to the compressed, misaligned instructions in the program memory. The method used in this work is to have only those addresses aligned in the memory which require random access to. The memory is arranged during compression in such a way that blocks of code which are to be executed in parallel, such as inner loop code, are tightly packed and freely misaligned, but occasionally an instruction which is the target of a control flow instruction re-aligns the execution to the memory boundary. This causes a need to be able to detect when the execution needs to stop within the misaligned memory areas due to an aligned instruction in the next memory address. In order to align an instruction to the current word, the previous memory word may contain padding information, which needs to be skipped. This can be communicated with a special *end of package bit* or *alignment bit* in the instruction. An example case of this is shown in Figure 3.1.

3.2 Program Code Compression

Compression is a procedure where input data is taken and reduced to a smaller representation with the help of algorithms designed for the data set. The algorithms form a compression model, which also tells how the data is uncompressed back to its original representation. For program code, the compression is used for reducing the amount of bits required to store or transmit data where it would be expensive. When the required storage or transmit phase is done using the compressed data, the

data is uncompressed back to its original size before usage. Common program code compression can be divided into the following approaches: NOP removal, dictionary-based compression, entropy encoding and instruction set re-encoding [26].

In the earlier days of computing, the availability of memory was much more limited and the usage of memory bytes was planned out carefully. Today, the concept of saving memory bits on hardware is just as important despite the increased availability of transistors, because low-power design has become necessary especially for battery-powered, embedded devices. Initially program compression methods were devised for the early single-issue *Reduced Instruction-Set Computers* (RISC) and *Complex Instruction-Set Computers* (CISC), but nowadays program compression is commonplace and is done for many computer architectures, even for the fairly complex multi-issue parallel processor architectures such as VLIW and TTA. Compression can be very beneficial for these architectures, as the very long instructions which make up the program code consume a lot of power.

Drawing the line between *encoding* and *compression* is complicated because the two concepts are tightly woven together. In this thesis, the distinction between encoding and compression is made by looking at their relationship with the processor: Compression is something that processes the data that is represented in bits. Even if the compression model defines how the data is to be unpacked on hardware, it does not communicate directly with the processor by means of instructions or bit toggling. Encoding is required to express instruction formats and compression models in machine language to the processor. For example, the instructions of an architecture can be *re-encoded* to a new format, which is used to compress the program code to a smaller size. Compression and encoding often go hand-in-hand as the compression model chosen for an architecture largely depends on the characteristics of the ISA, that is, the encoding and identifiable bit patterns of instructions play an important role in the selection of the compression model.

In practice, the compression of program code is done during compile-time by software and decompression is done during run-time somewhere in the processor's pipeline. Because the compression is done at compile-time, the speed at that stage is not an issue. However because the decompression is done by the hardware, careful consideration is required on where in the architecture's pipeline to perform the decompression as it affects the performance of the processor. Depending on the architecture, the placement of the decompressor varies. Commonly, the decompressor is located somewhere between the program memory and the processor's decoder, but in a processor with an instruction cache, it can also reside pre- or post-cache.

In a TTA processor, the decompression can be done either between instruction fetch and decode stages in the pipeline, or be integrated directly in the decoder. The TTA pipeline was shown in Figure 2.2. The unit which handles the instruction

decompression also consumes some area and power, depending on the complexity of the decompression hardware. If the compressed instructions are all fixed length, they are cheaper to decompress. A lot more complex logic is required in the hardware if the compression leads into the use of variable length instructions, which are more expensive to handle.

3.2.1 Compression Advantages and Challenges

The main motivation behind code compression on embedded systems is to reduce the program code's size, and consequently, the power consumed by the minimal instruction memory. The trade-off is additional complexity in the hardware because of the decompression unit and the variable length instruction fetch unit, if the compression model is encoded in variable length format. This requires more chip area and additional design time, and extra power is consumed by the resulting hardware. The power saving benefit from the compression must outweigh the extra power consumption of the added hardware. Otherwise there is no benefit in making the processor support compressed instructions.

Design Complexity and Speed

Since VLIW and TTA architectures are static-scheduled architectures by nature, their hardware is relatively simple. This makes designing efficient low-power hardware for variable length instruction challenging, as a poorly structured fetch and decompression units will consume a large proportion of the processor's area and may impact its performance. It is unavoidable that a variable length instruction fetch and decompression is more complex than its fixed length counterpart, because multiplexer and shifter logic structures are required for the handling of variable length instructions. Buffers are also required to store words read from the program memory, so the fetch unit can splice the instructions from the unaligned memory bits without interruptions in the execution.

Not only does the logic bloat the processor, but it may also require its own pipeline stage if the decompression scheme is particularly convoluted. In a compression model that is conveyed with fixed length instructions, the decompression may be simple enough to be solved in the same cycle as instruction fetch or some other procedure without affecting the processor's clock speed. The decompression of a complex variable length instruction can require a clock cycle dedicated to it unless compromising the processor's clock speed is allowed.

In order to reduce the complexity of the fetch and decompression hardware, some constraints should be set in the implementation. It should be considered if the instruction widths and hardware structures can be restricted to power of two values

inside the processor to minimize the logic. Some examples of design aspects to constrain are the maximum and minimum instruction lengths.

Power Consumption

The most important question is whether the compression helps save power in the system or not, when the memory power consumption is pitted against the various components required by the decompression hardware. There are a multitude of aspects to consider for power consumption, such as what kind of power consumption it is (dynamic, static), what type of transistors are used in the memory and on the rest of the design (low or high power) and so on. These aspects will have to be explored to understand how the power consumption is affected.

The power consumption of a circuit is divided into two categories: *dynamic power* and *static power*. The majority of the power dissipated in an integrated circuit is due to dynamic activity: net switching power, internal cell power and short-circuit power during logic transitions in the *Complementary Metal-Oxide-Semiconductor* (CMOS) transistors [1]. But the proportion of static power, also called *leakage power*, is approaching half of all power consumed in an ASIC as the contemporary technology node libraries keep advancing towards ever smaller line widths [2].

The largest share of leakage is *subthreshold leakage*, which is the drain-to-source leakage current when the CMOS transistor is off, also known as the weak inversion mode, and increases as the transistor's gate's threshold voltage decreases in each new, smaller transistor technology. The second largest type of leakage current is *gate leakage*, which is leakage through the gate terminal of a transistor. However, gate leakage became a less significant component when high-k + metal gate transistors were introduced for 45 nm technology, which reduced the gate leakage current by over 90% [27]. In several power measurement benchmarks done in [1], the average percentage of leakage power out of all power consumed in an IC is approximately 42% for the 22 nm technology node. The measurements also show that for the 22 nm node, gate leakage is less than 10% of the total leakage power.

Not all transistor types have equal dynamic and leakage power properties. The synthesis of an IC can take into consideration which parts of the hardware require higher performing transistors than others, and thus transistors with higher voltages and worse leakage properties are assigned on the critical path of the chip. The same can't be done for the memory used in the system: the entire memory module is manufactured from the same type of cells, which use either high performance or low standby power type transistors. HP SRAM reaches higher clock speeds but a substantial amount of current leakage is present [3]. The slower LSTP SRAM is used to minimize leakage, but the memory cells have higher on-currents, consuming more dynamic power as a trade-off.

Whichever SRAM cell technology is used, there are benefits in reducing the memory size with code compression. Less power is dissipated due to leakage current in a smaller memory module if HP SRAM is used, while less dynamic power is used on expensive memory read accesses in LSTP SRAM if multiple instructions can be read per cycle when variable length instructions are in use. Memory accesses become cheaper power-wise if the width or length of the memory can be reduced with a compression approach [6]. Finally, by restricting the clock speed of the design, the synthesis tool can allocate more low power transistors and create less complex logic structures for reduced dynamic power consumption.

3.2.2 NOP Removal Compression for TTA

No-operations are very common in serial code, which is compiled for a TTA machine providing ILP. Especially in large machines with many buses and function units, during operations which control the flow of the program, many of the FUs may not be able to be used for calculation as the flow changes. Changing from one tight parallel calculation loop to another is one such instance, another is pipeline flushing after a jump instruction. In addition, larger machines can be more difficult to optimize for a task, which results in NOPs on the buses.

For some TTA processors designed with TCE, measurements showed that application specific processors can still have up to 50% NOP moves out of all moves, whereas processors which were designed to run several different benchmarks could have up to 80% NOP moves in all of the program code for some of the programs due to program-to-architecture incompatibility.

The bit pattern for a TTA NOP in an instruction's move slot is always the same per program per move slot, still consuming the same amount of program memory like any other move in the same slot. The NOP removal method chosen depends on which way precisely the NOPs are arranged in the code. Three different NOP arrangements can be identified: single NOP, vertical NOPs and horizontal NOPs [13]. Figure 3.2 explains each of these configurations.

Vertical NOPs appear in the code when the compiler is unable to fill pipeline delay slots with useful operations, such as when flushing the buffer after a jump instruction. A NOP is seen on all buses for more than one cycle. Removing vertical NOPs can be done with a multicycle NOP instruction, which fills the pipeline with NOPs for several cycles. Multicycle NOP removal is implemented in EPIC [19], for example, which is further detailed in Chapter 4.2.1. For full instruction length multicycle NOP instructions to be cost-effective, there would have to be at least two consecutive vertical NOPs. Multicycle NOP instructions are not implemented in this work, but removing vertical NOPs can still be useful as they were measured to occupy 5–10% of program code on some TTA processor-application configurations.

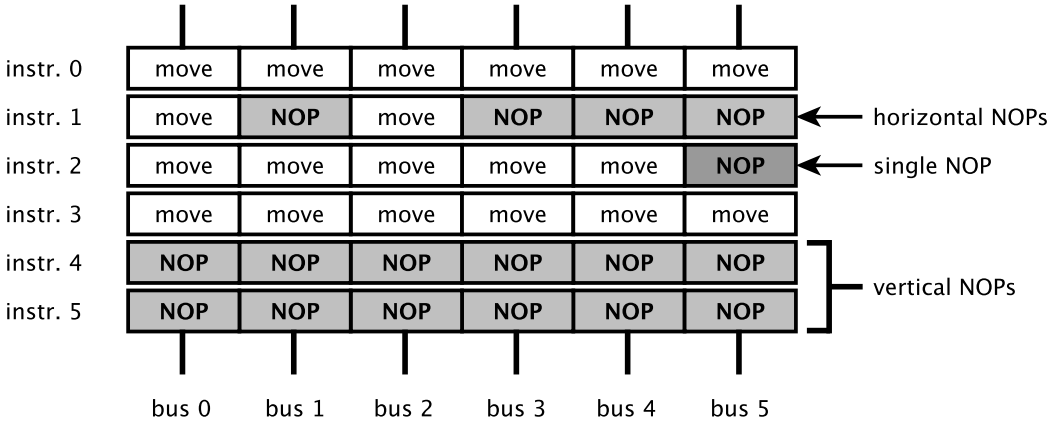

Figure 3.2: Examples of different NOP placements in program code, in relation to the instructions and buses. A single NOP can reside alone in an instruction on any bus. Vertical NOPs mean a situation, where all of the buses contain a NOP move for more than one cycle. Horizontal NOPs are multiple NOP operations within one instruction, but not necessarily in consecutive issue slots.

Horizontal NOPs appear when the compiler is not able to use all the buses within one cycle, having to allocate a NOP move instead of useful operations for several buses. A few methods have been commonly used in the past to remove horizontal NOPs, but ultimately it requires changing the instruction encoding for the processor, some of which can lead to variable length instructions. In this work, instruction templates are defined to represent shorter instructions, where the template tells which buses in the machine have had NOP moves for the instruction. This leads to a variable length instruction encoding. The NOPs are later filled back in according to the template bits found in the instruction encoding. This method is detailed in Chapter 5.1.

4. CODE COMPRESSION IMPLEMENTATIONS

Many different program compression methods for VLIW architectures have been reviewed in the dissertation [26] and three new compression approaches for TTA are introduced in the same publication. This chapter presents the main ideas of the previous TTA compression methods and a few of the VLIW compression ideas, which inspired the architectural solution for the proposed variable length instruction encoding on TTA. In addition, the encoding of a few well-known commercial architectures are reviewed, which either have already influenced or may affect the future direction of variable length instruction encoding on TTA.

4.1 TTA Compression Methods

Three experimental TTA instruction compression methods have been already measured and partially implemented in [26]. Three interesting metrics for each method are compression ratio, chip area used and amount of power saved. Another thing of especial note is whether the method requires fixed length or variable length instructions to decompress. The results put variable length techniques in a bad light because of the increased area cost and multiplied power consumption, but these may be overcome with more cost-effective units to handle the fetch and decompression of variable length instructions.

The compression ratio in the following compression approaches represents the program size after compression, meaning smaller is better.

4.1.1 Dictionary Compression

Dictionary-based compression methods all work on the same principle and have been studied not only for program code compression, but for text compression and picture compression as well. For program code compression, unique bit patterns are sought for in the code, and each such pattern is stored into a dictionary. This is always done during compile-time. In the dictionary, each entry is given a dictionary index, that is, a number represented as a fixed length bit pattern. The program code is thus compressed where-ever a unique bit pattern is successfully replaced with a shorter index. During decompression, which is usually done in a separate hardware unit during run-time, the indices are expanded to their corresponding bit patterns found in the dictionary.

What affects the compression ratio of the dictionary compression is the granularity used when searching for bit patterns, and the availability of convenient patterns. If the entire TTA instruction were to be used as the unique bit pattern to be replaced, it would lead to poor compression ratio in a complex-ISA processor since the number of different bit patterns is large. Each different instruction would get its own index in the dictionary. This would lead into a very large dictionary having to be stored in the decompression unit. The other levels of granularity for a TTA instruction which have been measured are the horizontal move level, vertical move level, horizontal ID-field level and vertical ID-field level. Horizontal move slot granularity means that all the moves within an instruction are considered as a single unique bit pattern to add to the dictionary. Vertical move slot granularity considers each move slot within an instruction as a unique bit pattern, which means there are as many dictionaries created as there are move slots. Finally, ID-field level divides each move slot to a guard-pattern, source-pattern and a destination-pattern, similarly in horizontal and vertical configurations as for the move slots. At the move slot and ID-field levels, one additional dictionary is created for the LIMM instructions, which span one or multiple move slots.

On the whole, the best compression ratio of 52.5% is reached with vertical move slot level, 62.5% at horizontal move slot level, 62.8% at vertical ID-field level, 68.9% at horizontal ID-field level and the worst 76.3% at the instruction level. An important result to note is that the best area and power savings are achieved at the full instruction granularity, which has the worst compression ratio. When its dictionary is synthesized on an ASIC technology as standard cells, the large table of bit patterns is optimized well by the logic synthesis tool, whereas with vertical move slot level the resulting several dictionaries are harder to reduce in size and the instructions have to be pieced together from several dictionary entries. Even with a better compression ratio, vertical move slot level's program memory becomes approximately 3 times larger than that of instruction level's due to multiple dictionaries.

Dictionary-based compression is generally a fixed length solution, but can result in variable length instructions if combined with dictionary size optimization or vector quantization. The latter two methods can be utilized if the support for variable length instructions is added to TTA. The dictionary-based compression approach's area reduction can reach 75% and power consumption can be reduced by up to 66%. The control logic required for the dictionary is fairly simple if it's integrated into the decoder. The downside of dictionary-based compression can be that reprogrammability of the system is lost unless both the dictionary and the program memory are stored on SRAM.

4.1.2 Huffman Coding

Huffman coding is a commonly used entropy encoding method for compressing patterns such as program code. The idea is to find the most common bit patterns in the code and assign them a short code word created out of bits, where the allocation is based on how frequently the pattern appears. The more common the pattern is, the shorter code word it is given. In addition, none of the code words are a prefix of another. This results in a table similar to that of dictionary compression, but the entries are in most common order and are accessed with variable length indices.

Huffman coding's granularity levels for TTA are the same as dictionary compression's: instruction level, move slot level and ID-field level. The compression ratios achieved were worse than dictionary-based compression's at instruction level: 81.9% on average. Vertical move slot approach reached a much better 43.1% and horizontal 53.7%. Even better still, ID-field compression reached 40.9% with vertical Huffman coding targets and 49.4% with horizontal.

Looking at the compression ratios, Huffman coding seems like an attractive solution, but the control logic required to handle variable length code word decompression bloats the TTA processor's area significantly. Exact measurements haven't been presented for how large the control logic and program memory together would be for this compression method, but it can be interpreted from the other results in [26] that the control logic alone becomes larger than the combined program memory and logic area requirements of dictionary-based compression solutions. Power results reveal the same trend: dictionary-based compression's simple control logic only consumes approximately 11 mW on a 130 nm ASIC technology, while the control logic for variable length instructions requires at least 40 mW.

4.1.3 Compression Based on Instruction Templates

The instruction template-based compression approach is a NOP removal technique for TTA which results in a variable length instruction format. It has been evaluated by Heikkinen for TTA in [26], but originates from work by Aditya et. al. for EPIC in [28]. The principle of the compression method in this thesis is the same as implemented Aditya's work, shown later in Figure 5.1. Several templates, an amount which is optimally a power of two, are defined based on the architecture and the program that has been compiled for the architecture. The template is indicated by a set of bits that is affixed to the MSB-end of the instruction and denotes which buses contain normal moves and which NOPs. The instructions can thus be shortened when written into the program memory during compile-time by discarding the bits for NOPs from move fields which are known to be null transports because of the template. The instructions are expanded again to contain the NOP

bits by interpreting the template in the decompressor during run-time. The best templates for each application are chosen by looping through the once-compiled program code and finding which buses most commonly have NOPs.

With 16 defined templates, the instruction template-based compression reaches an average compression ratio of approximately 50%. As more templates are defined, this counts as overhead in the program memory because the template has to be added to each instruction, but ultimately a ratio of 46.5% is achieved when using more than 32 templates. The results have been gathered from three different processor configurations, each of which have been designed to match the benchmark applications sufficiently.

The power and area results in this method are undesirable because of control logic explosion in the decompressor. The more templates there are, the more complex the system becomes. On 16 templates defined, while a decent compression ratio of the program memory was reached, the total area of the program memory and control logic combined increases by 80% compared to a design without any NOP removal done. The power consumed increases by 152%. Optimization to the fetch and decompression of the variable length instructions must be done to make this approach a viable solution for code compression.

4.2 Commercial Architectures

Many ISAs have improved over time by re-encoding or optimizing existing architectures into new variable length formats. Following is a short list of some well-known architectures which use NOP removal methods or variable length instructions.

4.2.1 Explicitly Parallel Instruction Computing

Schlansker and Rau introduced EPIC [19], an architectural design philosophy which was evolved from VLIW to increase ILP in processors. EPIC is more commonly known as IA-64 or Intel Itanium architecture today. Similar to VLIW, the detection of dependencies is static and ILP is exploited already in the compiler stage. But EPIC allows dynamic scheduling and offers code compatibility across generations. However, often a software recompilation is required for better performance when changing the target processor for the program.

EPIC's *MultiOp* instruction format describes multiple operations in one instruction that are intended to be issued simultaneously. Also so-called *MultiOp-S* semantics allow instructions which consist of *chunks* to have the chunks to be issued in parallel or sequentially, while their contents must be issued simultaneously. EPIC also contains two different variable length encoding schemes to eliminate explicit NOPs from the program code: *MultiTemplate* and *VariOp*, in addition to its fixed

length MultiOp instruction format, which contains a field for how many full NOPs are to be issued after the current instruction.

MultiTemplate instruction format involves the use of templates, each of which defines a subset of function units to target with the operations of the variable length instruction. The rest of the FUs are implicitly provided with a NOP. The shorter templates can be utilized during control code, while wider templates are used when more FUs are targeted with operations. Only a limited set of templates are defined based on NOP frequency to prevent the fetch and decompression logic from growing out of feasible proportions.

The VariOp instruction format is different, as it permits any subset of operation slots to be included within any instruction up to the maximum FU amount. Each operation is explicitly targeted to a FU and the remaining empty operation fields are implicitly filled with NOPs on a per instruction basis.

4.2.2 ARM Thumb

ARM's Thumb architecture [29] was introduced in the year 1995 to address the poor code density that was common for 32-bit RISC processors in the mid-90s, as memory was often the most expensive component for embedded devices of that age. A subset of 36 instructions of some of the most common and simple 32-bit instructions was re-encoded into 16-bit-wide instructions, typically net resulting in 70% compression ratio in comparison to regular ARM code. As a trade-off, the instructions could only use 8 registers in the 16-bit mode.

Prior to the decoding of ARM instructions, the Thumb instructions are translated from the 16-bit representation to their 32-bit equivalents on-the-fly with an integrated decompression unit. In practice, a state switch had to be made inside the processor by executing a specific instruction which would write to a status register. This way the information about the instruction's length would not have to be affixed to each instruction as a template, saving on program memory bits.

The instruction memory width could be 8, 16 or 32 bits. If the program was such that it contained chiefly the shorter 16-bit-long Thumb instructions, then using a 16-bit wide memory was beneficial for performance purposes. Regular ARM instructions would require two clock cycles to fetch in this case. But a 32-bit memory could be used as well, where two Thumb instructions would be read in at a time and the input latches worked as a one-instruction-long prefetch buffer. Regardless, the control unit of the Thumb processor stays fixed width and fairly simple, since there are less problems with memory misalignment with only two instruction sizes which are consecutive powers-of-two.

Applying this idea for a TTA processor, nearby power-of-two instruction widths can be used to simplify the processor hardware. In addition, when a memory word

is read which contains multiple smaller instructions, the fetch buffer works as a prefetch buffer, saving on power due to memory reads as the instructions within the buffer are dispatched.

4.2.3 x86

The x86 is a popular backwards compatible dynamic architecture family, which is based on Intel's 16-bit 8086 CPU from 1978. It has since then been extended many times to become a CISC architecture with variable length instructions, which range from 16 to 128 bits long [10]. The extensions have provided more functionality and new data types most often for *Single Instruction Multiple Data* (SIMD) calculation needs in the processor. Because of the ISA's complexity and variety, it is mostly used on personal computers which use a lot of processing power, instead of very low power embedded systems.

The x86's 64-bit version, x86-64, has a fairly good code density due to its CISC design, as is detailed in [30]. It is compared to multiple other CISCs and RISCs, faring well against even some embedded architectures such as the Thumb in a few code density benchmarks. While large multi-byte instructions are available, much can be achieved with shorter instructions, which compilers also tend to use. The same study also lists IA-64 in the density benchmarks, which gets very poor results due to many NOPs being generated in the code despite template-based NOP removal. The benchmark, however, isn't optimal for a VLIW-style architecture as the code is said not to be inherently parallel, but mostly serial code. Two things here are worth noting: First, even if an efficient template-based NOP removal method is used in a static multi-issue architecture such as IA-64, the density still suffers if parallelism can't be taken advantage of in the application. Second, while the x86-64 has good code density, its decoder is very complex because of the ISA's multiplicity, thus isn't appropriate for very low power embedded systems.

4.3 Other Compression Methods

VLIW architectures are the closest relative to TTA in the field of ISAs, which is why it's a good place to look for compression methods to utilize for TTA. Below is one approach which is of interest because of its similarity to template compression. Most non-commercial approaches to VLIW code compression by NOP removal result in variable length instructions. Many of these compressed encoding mechanisms have attained good compression ratios, but often changes to chip area and power consumption have been left out of the results. That is not to say they are all poor in that regard, but that easily becomes the assumption unless reported otherwise, after seeing the results of instruction template-based compression on TTA.

4.3.1 Tagged VLIW in DSPs

Weiss and Fettweis propose a *Tagged VLIW* (TVLIW) ISA to be used in DSPs in [31], which performs NOP removal by only using large instructions when necessary. In the TVLIW approach, one VLIW instruction is constructed from one or more TVLIW-instructions. One TVLIW instruction contains a class field, two tag fields and two functional unit instruction words, allowing the control of any two FUs in the machine. The remaining FUs are implicitly supplied NOPs. The tag fields specify which FUs are targeted and the class field tells how many TVLIW instructions are to be combined. If the functionality of more than two FUs are desired, the larger VLIW instruction is assembled from several TVLIW instructions which may take several clock cycles.

The advantage that TVLIW brings is that control flow instructions between loops can be carried out with a single shorter instruction. During in-loop execution, where more FUs are often needed for the part of code that works well in parallel, the VLIW instructions are assembled from more than one TVLIW instruction. The decoding logic is said to be simple and a loop cache can be used to necessitate the slow assembly of full VLIW instructions for only the first iteration of a loop. The mechanism of supplying unused FUs with NOPs is something TTA could use to save on program memory bits. The loop cache near the fetch unit in TVLIW is also a good idea for a future enhancement. It introduces a possibility to save power on memory reads by containing the recent loop instructions in a cheaply and quickly accessible cache.

4.3.2 Heads-and-Tails

Heads-And-Tails (HAT) format [32] by Pan and Asanović is one of many instruction re-encodings, which works on the principle of adding new control bits to the instruction and grouping several instructions together. Its purpose is to enable better parallel fetch and decode for variable length instruction encodings used in dynamic architectures, such as compact-RISC and CISC.

An example of the HAT instruction encoding is shown in Figure 4.1. The format splits each instruction into a fixed length head and a variable length tail, which are contained in a larger fixed length instruction bundle along with other heads and tails. Smaller instructions can be contained within only the heads, so a tail isn't necessary for each head in the bundle. The corresponding tails are located by information in the heads. Heads are always packed together in program execution order and the tails are packed together in reverse order. The idea is that the bundle itself is always aligned in the memory and the heads of each instruction are the same fixed amount of bits apart, making cache alignment and instruction decoding easier, while the

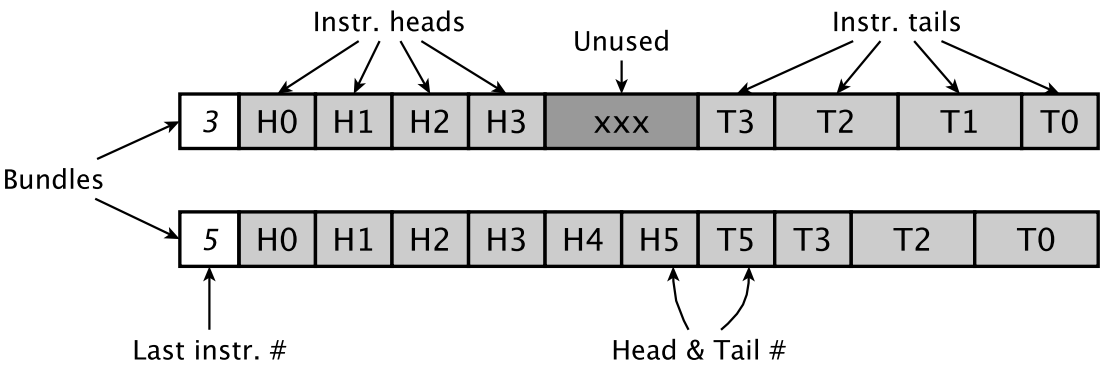

Figure 4.1: Heads-and-Tails instruction examples.

variable length tails improve code density. Depending on how HAT is used, the tail complexity can vary a lot and determines the minimum and maximum bundle sizes. For example, a tail can be completely omitted, or it can contain 6 immediate value fields.

Experimental results from HAT's effectiveness are gathered by re-encoding a MIPS RISC [33] instruction set in a variable length HAT scheme. This MIPS-HAT was evaluated with 128-bit and 256-bit instruction bundles. The former bundle size contains up to 8 instructions and latter up to 16. Compression ratios of 78.5% and 75.5% are reached, respectively. Area, power and performance results are not disclosed. HAT could be adapted for TTA to handle NOP removal or offer a subset encoding mechanism for non-NOP moves, and could be efficient with the simple instruction fetch it requires. However, a TTA processor's move decoding within bundles would become complex and costly because of the need to find each tail every cycle. Also some code density is lost on the padding bits in the middle of the instruction format, as each bundle is aligned in the memory.

5. DESIGN AND IMPLEMENTATION

This chapter is divided into three parts, where the first one explains the compression method which is used in TCE's TTA implementation for NOP removal. The second section is a documentation of the fetch units and details the decompression routine required for instruction template-based compression. The last section talks about the software integration of the new hardware designs into the existing TCE toolset.

5.1 Instruction Template-Based Compression

The compression method utilized in this thesis is instruction template-based NOP removal compression, which uses variable length instructions. The template approach originates from work done in [28]. This compression method can be used to remove horizontal NOPs efficiently. It re-encodes the processor's instruction set by modifying the behavior of TTA's instruction control field and adds an alignment bit for dealing with random access support. The template field is used for defining instruction formats which contain information for only a subset of the available move slots in the processor. A template defines which move slots are included in the instruction format, therefore the instruction's size is also tied to the template. The move slots that are left out of the selection of a template are implicitly assigned NOPs in the decoding stage. Such move slots become *NOP slots* in that template.

An example of template selection and horizontal NOP removal for a 5-bus TTA processor is displayed in Figure 5.1. In this example, a large amount of NOPs are seen in four instructions. The alignment bit is omitted from the example for clarity. Two new instruction formats are assigned to the templates *'10'* and *'11'*, which only use the buses A, B and D, E. The rest of the buses in these two formats are considered as NOP slots. If NOP moves are seen in the NOP slots, they are removed from the instruction. These two NOP removal templates can be used in three instructions to remove a majority of the NOP operations in the program code.

As seen in the example, the template previously used only for long immediate unit selection was changed to be used for NOP removal as well. This means that in addition to the necessary base template which defines a move for each bus, at least one template is required by immediate unit selection. Due to the binary representation of templates in the template field, the amount of templates for each machine is optimally a power-of-two number, but can be less for reduced decoder complexity.

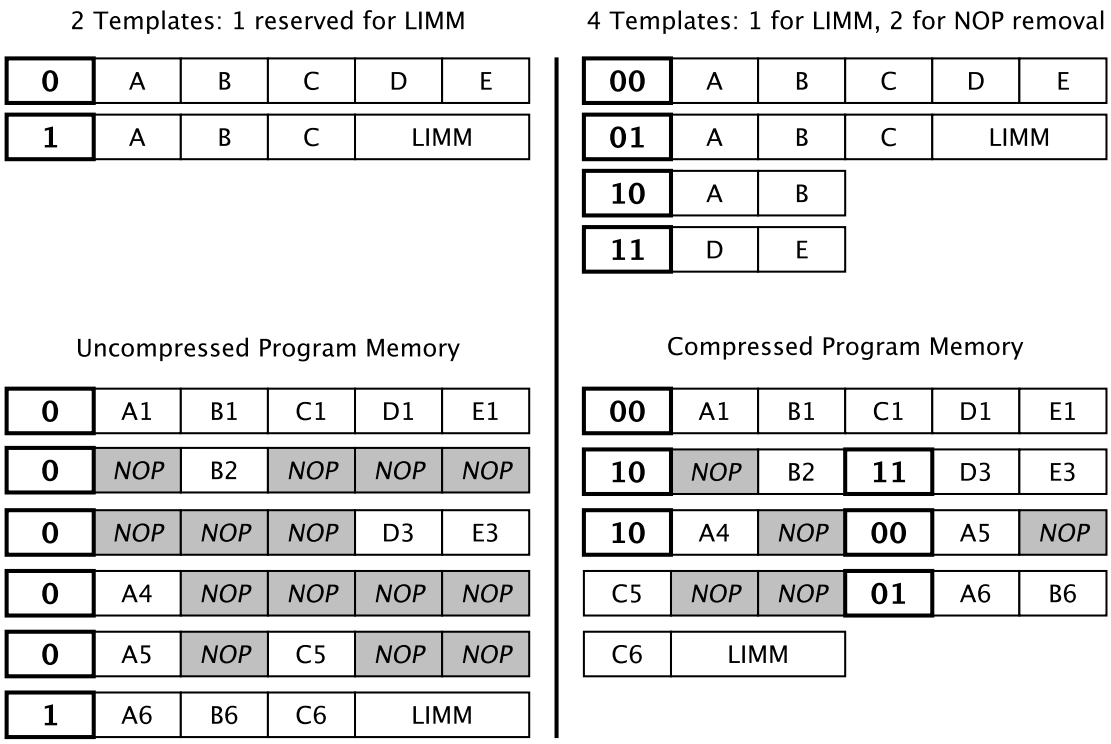

Figure 5.1: A short program before (left) and after (right) assigning two new instruction formats, which define two move slots to be used out of the five in the processor. Most of the NOP operations are removed by using the shorter instruction formats in the 2nd, 3rd, and 4th instruction. One extra bit is added to every instruction to be able to denote the two new templates to the processor.

The selection of templates can be done by first compiling the source code once without the use of templates, then looping through the disassembled program format to find the move slots which most commonly have NOPs assigned. Good NOP slot patterns to choose are those which reduce the instruction size to or just below a power of two value for less hardware generated for decoding. It's also important to choose such NOP slot combinations for the templates where the NOP slots do not overlap, as one template can be a superset of another. A template which can be beneficial to choose is one where all the instruction's move slots are NOP slots, because some full NOP instructions are often present in jump delay slots.

The actual use of the templates for compression happens during program scheduling at compile-time. Each instruction is attempted to match to the list of defined templates starting with the template with most NOP slots used. This may not lead to the best compression ratio because slots can have varying widths in TTA, but is a good starting point. If an instruction can be matched with a NOP slot template, the given template is assigned to the instruction and the bits for each of the matched NOP moves are removed by a compressor during program image generation. The removal of bits in the instructions affects jump address mapping to the misaligned

program memory, which is the random access support problem mentioned earlier.

Random access support on TCE's TTA was addressed partially in the compiler and partially inside the fetch unit. Jumps and calls are supported by having all their target instructions in the program code to be aligned at memory addresses. The code becomes divided into blocks which are mostly misaligned due to variable length instructions, but occasionally aligned again due to a jump target. The issue with sudden alignment is that the instruction prior to an aligned instruction may contain redundant information, *padding bits*, which needs to be disregarded. The padding bits are detected by appending an alignment bit to the MSB-end of each instruction, indicating whether the current instruction contains padding bits in the memory word after the actual instruction bits. This bit is *'1'* if padding bits exist.

The instruction template field is read during run-time in the decompressor and the instruction is pieced back together from the variable length representation to the processor's maximum instruction length by re-inserting the missing NOP bits to the NOP slots. Because of the operation's simplicity, the decompression is done in the decoder during the same clock cycle as the regular dismembering of instructions, instead of in a dedicated decompressor unit. The complexity of the re-assembly depends on the amount of slots in the processor, number of templates, maximum instruction width and the bus widths.

5.2 Hardware Implementation

Two different alternatives for fetching variable length instructions were designed to be used in TCE-designed TTAs: a *Ring Buffer* (RB) fetch and *Shift Register* (SR) fetch. The former is a minimalist design which aims directly for the least logic consumption, and thus for least power consumption. The latter addresses some of RB's complex control logic requirements by adding more buffer space. An automatically generated decompression structure was also integrated into the processor's decoder.

In order to reduce the fetch and decoder complexity, the smallest variable length instruction size allowed in the processor is constrained to a certain *quantum* (q) size. The q and *maximum instruction size* (I_{max}) define the size of the multiplexer and shifter networks generated. This q can be increased from *instruction template bit field width* (ITF_w) + *alignment bit width* (a_w) to I_{max} for the least logic in the processor, but worst decompression ratio of the instruction template compression. In effect, a_w is always 1 and ITF_w depends on the amount of templates defined. If q equals I_{max}, the instructions become fixed length. The q and I_{max} should be power-of-two values for minimal logic increase. Furthermore, the *instruction memory width* ($Imem_w$) was set to equal I_{max} for ease of the fetch units' design. This means that at least one instruction's worth bits are fetched from the memory per read, but not necessarily all the bits of an entire instruction due to memory misalignment.

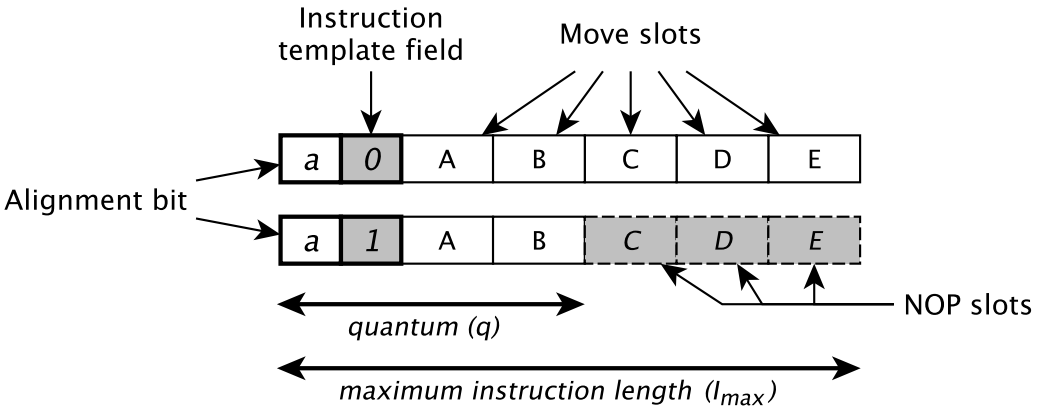

Figure 5.2: An example of the variable length instruction format used in this work. There are two templates in this example machine: '0' and '1'. The 0-template is a base template with all move slots in use. 1-template is a NOP removal template which has assigned C, D and E slots as NOP slots. An *alignment bit* is used to tell whether the next instruction is a jump target. The *instruction template field* defines which long immediate or NOP removal template the instruction belongs to. The *quantum* points out the smallest variable length instruction size in the machine. The *maximum instruction length* is the length of the widest instruction, which equals the memory width in the current implementation.

An example of the final instruction format based on these restrictions and the instruction template-based compression is shown in Figure 5.2.

5.2.1 Ring Buffer Fetch

This section first describes the ring buffer design in general. The following subsections are a documentation of the RB's VHDL processes on a conceptual level. In addition, some important other functionality that is decentralized within multiple processes is documented separately.

The RB design started from the requirement that at minimum 2 I_{max}-length instructions must fit into the buffer for continuous execution without needing to lock the processor occasionally. Minimal logic and register usage suggests minimal power consumption. The *ring buffer width* (Rb_w) requirement of $2I_{max}$ stems from the following restrictions: At minimum, only one instruction's bits per read cycle is fetched from the memory because $I_{max} = Imem_w$. Writing to the buffer must happen one cycle before reading (*dispatching*) from the buffer. In order not to overwrite any of the currently dispatching I_{max} instruction's bits with new bits, another I_{max} bits must fit into the buffer.

While the buffer may contain only one I_{max} instruction, it can instead contain a variable amount of instructions with widths between q and I_{max}. This means that the location of the next instruction to be dispatched varies and must be found. This is done by checking the current dispatching instruction's length from a *Look-Up*

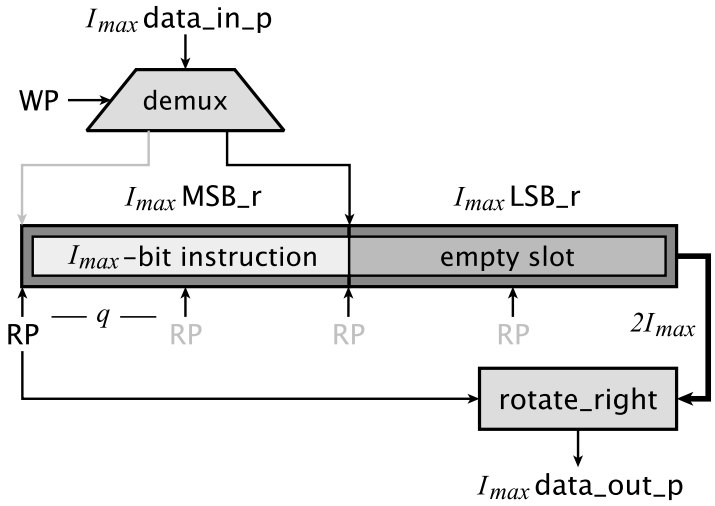

Figure 5.3: The structure of the ring buffer fetch unit for $I_{max} = 2q$. An instruction has been written to the MSB half on the previous cycle and is currently being read out of the buffer, indicated by the RP. WP is assigning the next memory word to the LSB half. The RB's contents are rotated with a *rot_r*-function by $RP + 1$ for output. Other possible RP locations are defined by the q. The data goes directly to fetch output port after rotating.

Table (LUT) from the index which the instruction template field bit value translates to. The LUT contains the length of each template and is constructed per processor.

Due to the constraint $I_{max} = Imem_w$, the buffer can be divided into half, that is, into *Most Significant Bits* (MSB) and *Least Significant Bits* (LSB). Only either half of the buffer is written into, so a *Write Pointer* (WP) is allocated to point either at the first bit of the MSB or LSB half. Consequently q limits the positions an instruction can be dispatched from in the buffer, which are pointed by a *Read Pointer* (RP). The term *current instruction* refers to the instruction RP is pointing at. Finally, as per the name of the buffer, an instruction to be dispatched can be located partly in the LSB's lower half and partly in the MSB's higher half. This complication is solved by writing the entire buffer's contents each cycle to a variable, which is rotated right by $RP + 1$ to align the instruction to be dispatched to the buffer's MSB. With all the RB's parts named, an example of its execution is shown in Figure 5.3.

Asynchronous Signals

Several important asynchronous signals need to be continuously up to date for the processes inside the fetch unit to work properly. The asynchronism in this case means that the signals are constantly updated instead of timed according to a discrete clock signal. Asynchronous signals update only when some other registered and clocked signals switch values, but glitching of signal values may still occur when asynchronous and synchronous signals are used together. Multiple-cycle hardware

designs are regularly made using finite state machines which have several states that update according to synchronous clock signals, but the fetch unit was not realistic to design with such means as it required several processes with asynchronous dependencies with each other. Asynchronous signals were required for obtaining information during the currently ongoing clock cycle.

The `RP_mode` signal contains the instruction template bits of the current instruction. These bits are translated to a `RP_instr_width` signal, which tells the *width of the current instruction* Ci_w using a LUT.

The `buffer_full` signal is a one-bit-signal which tells whether the buffer is currently full or whether a memory word will fit inside the buffer without overwriting undispatched bits. Because writing to the buffer happens with a delay of one clock cycle, this is performance-costly to calculate, but must be done to enable uninterrupted fetching and dispatching. The width of the current instruction must be determined to know if the RP moves forward far enough *this cycle* that a memory word could be written in the *next cycle*. The calculation causes a few delta cycles of delay and settles on the critical path inside the RB fetch design. The fullness must be calculated at the beginning of each cycle because of the buffer's narrowness. If the fullness were to be calculated into a register instead of an asynchronous signal, that is, with a delay of one cycle, a stall situation would occur if too long instructions had been written into the buffer. If the buffer was one I_{max} longer, this situation could be avoided and the calculation could be simplified.

The `padding_indicator` help signal informs whether the align bit is one or zero in the current instruction. A few processes inside the fetch unit depend on its value: RP calculation, buffer fullness, current instruction address upkeep and *return address* (RA) calculation.

Buffer Control Logic & Write Process

The fetch unit's main process called *fetch* is a synchronous process in charge of writing data to the ring buffer if fetching has been enabled for the cycle. It also updates the program counter register, initiates a synchronization countdown after a control flow instruction and controls the global lock signal which fans out from the fetch unit to other entities in the processor. The most complex task of this process is writing correctly to the ring buffer. The control logic whether to write to the lower or higher half of the buffer depends on buffer fullness, buffer emptiness and which parts of the buffer RP and WP are pointing at. The flowchart for selecting the buffer half to write to is displayed in Figure 5.4.

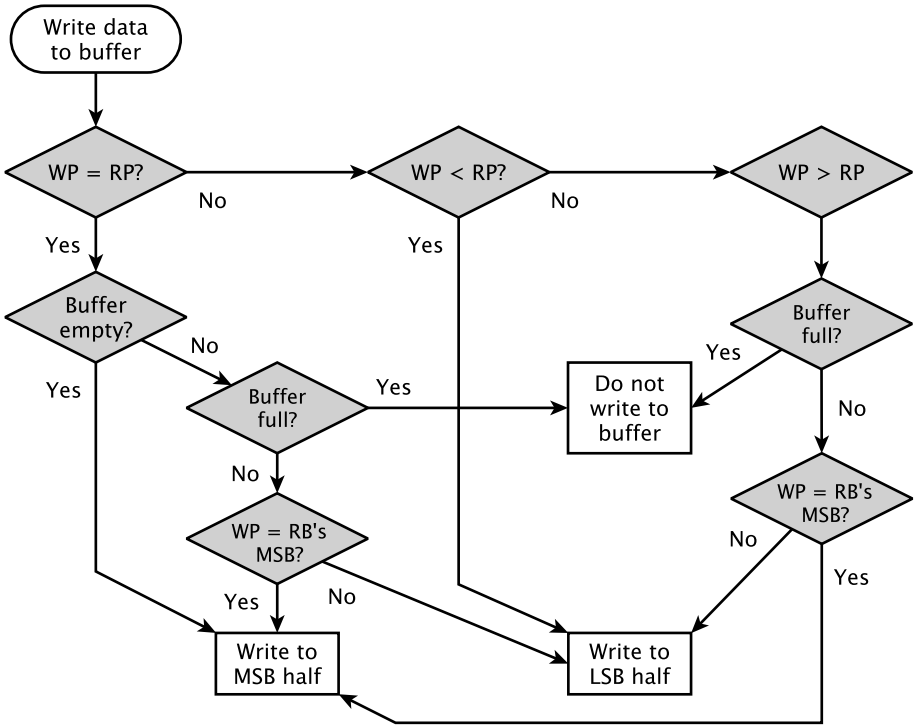

Figure 5.4: The flowchart to decide which half of the ring buffer to write to.

Buffer Output Process

The buffer output process is named *collect_output* and is sensitive to changes in the RP and the ring buffer's contents. It represents the *rotate_right* function in Figure 5.3. Its task is to save the entire ring buffer's contents to a variable, which is rotated right by $RP + 1$. This needs to be done so that the current instruction, even if split between the LSB and MSB of the buffer, becomes aligned at the buffer's MSB. Only the MSB half of the variable is moved to the fetch unit's output port every cycle. Every time an instruction smaller than I_{max} is dispatched, bits belonging to the upcoming instructions are extraneously dispatched as well, and need to be disregarded in the decompression phase. This is done automatically by the decompressor, because the template lengths are known globally.

Pointer Update Process

The pointer update process named *pointer_update* is a synchronous process which recalculates the values of the read pointer and write pointer each cycle. The WP value switches between the MSB half and the LSB half during every cycle that fetching is enabled. The RP value is more complex because it needs to wrap around the buffer correctly with varying instruction lengths, and as a special case it is updated differently if the alignment bit is present in the current instruction. The RP value calculation is presented in Algorithm 1.

Algorithm 1: Algorithm to determine RP value for the next clock cycle.

Data: alignment bit, read pointer value RP, maximum instruction size I_{max}, ring buffer width Rb_w, current instruction width Ci_w
Result: new RP value
Start;
if *alignment bit = 1* **then**
    if $RP - Ci_w >= I_{max} - 1$ **or** $RP - (Ci_w - 1) < 0$ **then**
        RP = RB LSB half's first bit;
    else
        RP = RB MSB half's first bit;
else
    if $RP < Ci_w$ **then**
        $RP = RP - Ci_w + Cb_w$;
    else
        $RP = RP - Ci_w$;

If the alignment bit is seen in the instruction, the RP always returns to the beginning of either buffer half depending on its current position. If the align bit is not present, the RP can wrap around the buffer if the current instruction is large enough and the RP resides in the lower buffer half.

Program Counter Update Process

The program counter update process named *sel_next_pc* updates whenever the signals `pc_load`, `pc_in` or `increased_pc` change. Its task is simple: when the `pc_load` signal becomes 1, that is, when a control flow instruction has been detected, the next program counter value is read from the `pc_in` signal instead of `increased_pc`. Instead of incrementing the PC inside the fetch unit, the next memory address comes from the processor's interconnection network, which is outside of the fetch unit. The value can arrive directly from an instruction, a register file or an FU.

Control Flow Instruction Handling

The correct execution of control flow instructions such as jump and call is complicated due to memory read latency, pipeline latency and a variable amount of instructions in the ring buffer. Inside the *fetch* process is a counter, which activates when the fetch unit's `pc_load` input port is seen with a value of 1, meaning that either a jump or call instruction has arrived to the processor's decoder. A chain of events is synchronized inside the fetch unit in different processes according to the counter value. The counter counts down from the pipeline delay amount of 3 down to 0. The chain of events is shown and explained in Figure 5.5.

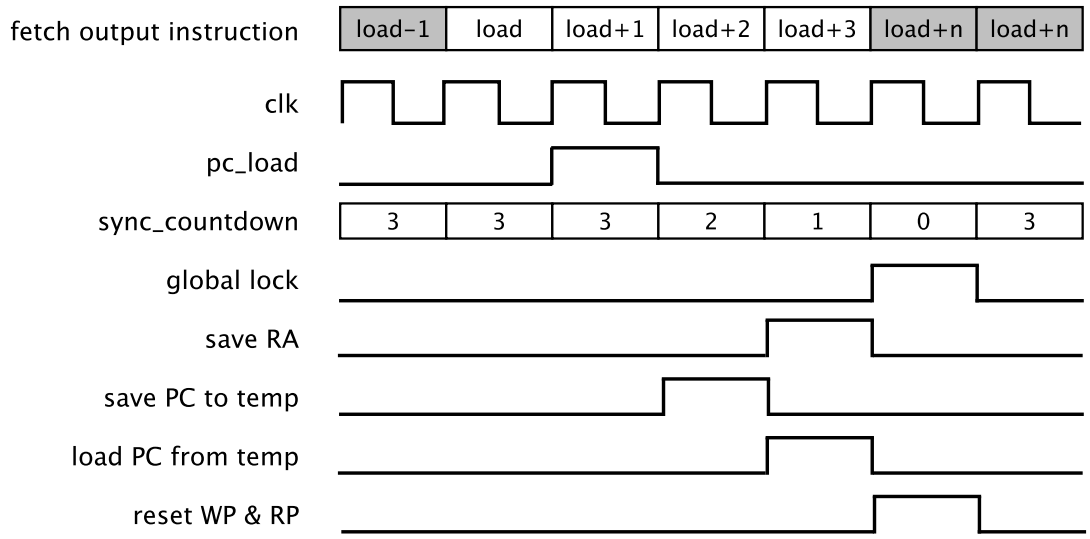

Figure 5.5: Synchronization time diagram according to `pc_load` signal. A cycle after a jump or call instruction has been dispatched, the `pc_load` signal comes from the decoder and a countdown is activated. The countdown value of 3 is the amount of pipeline delay slots. Three instructions after the *load* instruction must still be dispatched into the pipeline. At countdown = 2, the PC is stored to a register to anticipate staying at the same memory word for two cycles, one of which is during the global lock cycle, otherwise an instruction dispatch is missed. The PC is loaded at countdown = 1, and at this point the return address can be correctly calculated. Finally, the processor is globally locked at countdown = 0, during which RP and WP are reset. Normal instruction dispatching continues after the lock cycle from the new memory area where the processor jumped, indicated by *load+n* in the output. The same instruction is seen twice in the output due to the global lock cycle. The first one is automatically neglected in the processor.

Return Address Calculation Process

The return address is the address of an instruction where the program returns after a call or branch subroutine. After a call instruction, due to pipeline delay, the fetch unit still dispatches 3 instructions from the previous memory area before it starts fetching instructions from the new memory area. Since these instructions are variable length, the correct return address calculation is tricky.

The RA update process *update_ra_proc* is a synchronous process with two purposes: the process keeps track of the memory address of the current instruction RP points at in the buffer in an `addr_rp` signal and calculates the memory RA when a call-instruction is used in the processor. The `addr_rp` signal is incremented when all the bits of a memory word in the buffer have been dispatched and the RP is moved to the next instruction. With the help of this track keeping, we can calculate the correct RA at the clock cycle shown in Figure 5.5.

The RA calculation is complicated by occasional instruction alignment issues and padding bits. The RA varies depending on whether the instruction in the final delay slot before the call instruction crosses the memory word boundary to another

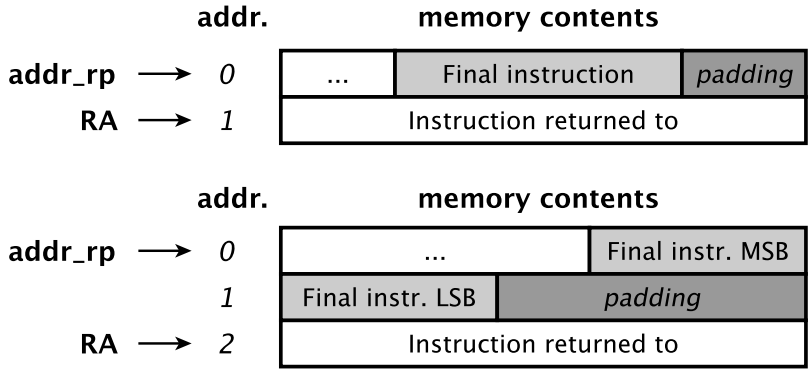

Figure 5.6: Return address variation depending on whether the final instruction to be dispatched before branching crosses the memory address or not.

memory word. This situation is depicted in Figure 5.6. Because of the return address's calculation complexity, a version of the ring buffer was designed which doesn't support call instructions. They are assumed to be replaced by the compiler into a jump and a register write operation. This also allows a small reduction in fetch logic.

5.2.2 Shift Register Fetch

The shift register fetch unit was designed to address the complex control logic and one-cycle synchronization lock cycle of the ring buffer fetch design. The major differences in the shift register design are: buffer size $Sr_w = 3I_{max}$ (instead of $2I_{max}$), simpler input/output logic and a shorter critical path thus better performance, but overall greater logic amount. Instead of a multiplexer structure to write into and read out of different places in the buffer, this design has a static write point and a shifter structure for output. Furthermore, SR doesn't support calculation of return address values. It expects calls to be translated to a jump and register write operations. Similarities in the designs are in the way align bits and templates are read from the current instruction to determine how the current instruction should be handled, and how the q value limits the complexity of the buffer. I_{max} equals $Imem_w$ in this design as well.

The contents of the buffer are continuously shifted left for storage by I_{max} so that the next I_{max} bits from the memory can be written into the buffer. Otherwise the incoming bits from the memory would overwrite bits already in the buffer. Simultaneously, every cycle one instruction is consumed from the buffer to the fetch unit's output. It can be imagined as a *First In First Out* structure which stays at equilibrium as long as I_{max}-length instructions are consumed. The buffer begins to fill up if shorter than I_{max} instructions are consumed, because the contents are shifted right each cycle by I_{max}, and the point where the current instruction is being

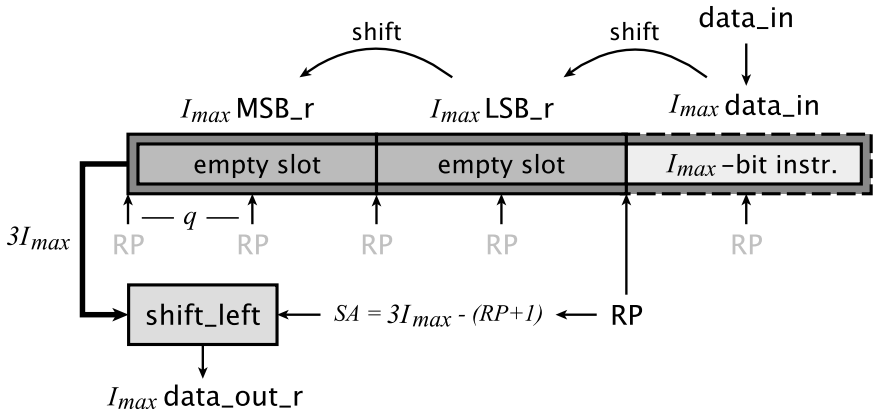

Figure 5.7: The shift register fetch structure for $I_{max} = 2q$. An instruction is seen in the data_in port and treated as a part of the buffer. The incoming instruction is I_{max} length and is shifted left to the MSB for output, indicated by the SA value, which is translated to a virtual RP value for easier visualization. Other possible SA values with respect to I_{max} are defined by q. The data propagates through a data_out register before output.

read from moves towards the MSB of the buffer. When the buffer is full, that is, the current instruction reaches the MSB slot, instructions are consumed until an I_{max}-size memory word can be written in again.

A read pointer can be imagined to point at an instruction within the buffer alike in RB, as shown in the execution example in Figure 5.7. The RP value is actually stored as a *Shift Amount* (SA) inside the logic to tell how much the contents need to be shifted left, so the current instruction can be aligned to the buffer's MSB in a variable for output, similar to the RB's output process. This shifting-for-output is represented as the *shift_left* function in the aforementioned figure.

A major difference in the SR fetch is how it uses its `data_in` port as part of the buffer. But since the `data_in` port cannot be directly routed to the `data_out` port because of fetch unit's designated 1-cycle pipeline latency, an I_{max}-length register is needed at the output. This results in SR design being larger and consuming more power than the RB design if they are designed with same I_{max} and q parameters.

The HDL written for the SR fetch is better structured than that of RB's. Also due to simpler control logic, most of the processes are easier to follow than RB's. Important asynchronous signals are the same in both fetch units, except that buffer fullness checking does not need a signal with a complex algorithm of its own in the SR design as its buffer is larger.

The SR's HDL process names are *pc_control_proc*, *rstlock_proc*, *lockcnt_proc*, *fetch_proc*, *shifter_proc*, *shift_pointer_proc* and *ra_calc_proc*. How the processes work is relatively simple to follow in the VHDL code itself, which has been well commented. They are not documented exhaustively here, as the design is subject to change in the near future according to the future work presented in Chapter 8.

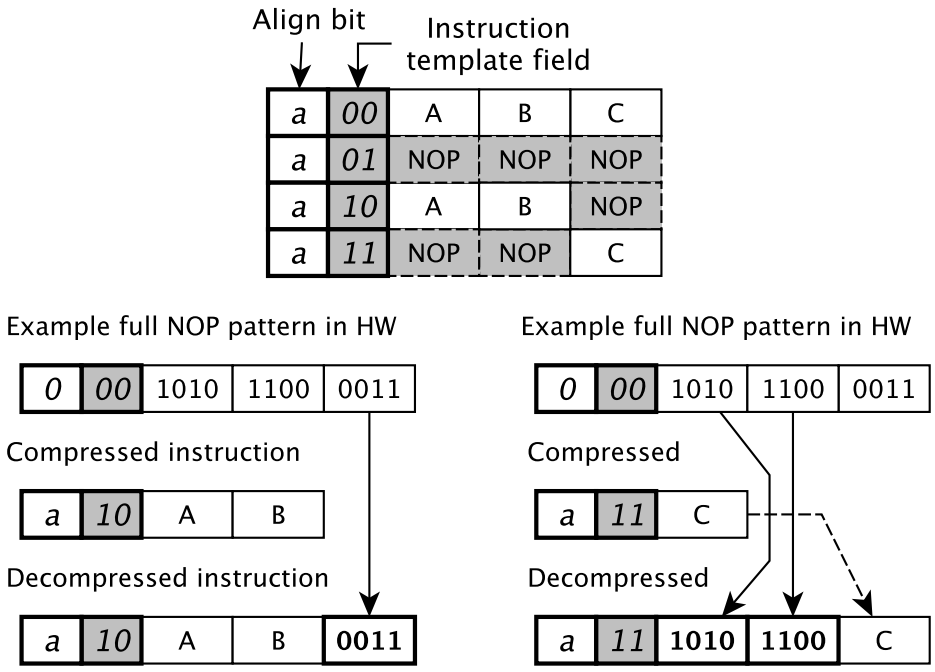

Figure 5.8: Two examples of instruction template-based decompression on hardware. The full NOP pattern, which is stored inside the decoder as a hard-wired constant, is used to fill in the missing NOP slots in the compressed instructions.

5.2.3 Template Decompression

The instruction template-based compression is decompressed in the decoder unit, which also decodes the instruction format for the TTA processor's interconnect on the same clock cycle. Two-template decompression was synthesized first on hardware as a trial to see how it affects the performance. It was determined from preliminary synthesis results that the decompression is a simple enough procedure that it wouldn't negatively impact the processor's clock speed even if done on the same cycle as decoding.

The decompression expands each variable length template instruction back into the full length instruction format of the machine. Missing NOP bits are added back for each template. To do this, a constant which contains the NOP bit pattern of every move slot, called a full NOP instruction, is stored in the decoder. A custom decompression hardware is generated per processor which decompresses each template separately, as shown in Figure 5.8. As seen from the example, the decompression procedure needs to fill in the expanded NOP operations into the correct places in the incoming compressed instruction, which varies per template. The complexity of this operation depends on how many move slots are in the instruction format and how fragmented the selection of the NOP slots has been for the templates. Move slots can also have varying widths with respect to each other.

6. INTEGRATION TO SOFTWARE TOOLSET

This chapter first gives an overview of the TTA-based Co-design Environment, which the variable length fetch units and instruction template-based compression's decompressor were integrated to for rapid processor prototyping and generation. Then the new features added to the toolset's programs are documented.

6.1 TTA-Based Co-Design Environment

TCE is a design toolset which provides an automated design flow for creating synthesizable TTA-processors, developed at the Tampere University of Technology [34] in Finland. The ability to create processors which contain a variable length instruction fetch unit is integrated into the TCE toolset as a part of this thesis. This requires modification to some of TCE's programs. Below is an overview of the main programs in the toolset that are used in synthesizable TTA-processor generation: *ProDe*, *ProGe*, *tcecc* and *PIG*. The programs and their relations to each other and to data formats used are displayed in Figure 6.1.

One of TCE's core tools is the *Processor Designer* (ProDe), a processor architecture design program with a graphical interface. This tool allows for creating new architectures and modifying previous ones by connecting FUs, register files and other units to buses. The processor's structural information is saved as an *Architecture Definition File* (ADF). An *Implementation Definition File* (IDF) can be saved as well, which contains information such as which function unit in the architecture is tied to which hardware implementation found in the *Hardware Database* (HDB). The IDF also lists which plugins are associated with the hardware generation and what special parameters are used in the generation, such as if bus tracing capabilities are desired in the resulting HDL. Moreover, the command-line program called *Processor Generator* (ProGe) can also be evoked from ProDe after the desired architecture has been created and function units have been selected.

The processor architecture designed by ProDe is converted into a synthesizable HDL environment using ProGe. In addition to the actual HDL files for all the hardware units in the architecture, a simple testbench is created along with compilation and simulation scripts for running the program contained in the memory images. Simulation scripts are created for the open-source HDL simulator GHDL [35] and the commercial simulator ModelSim [36].

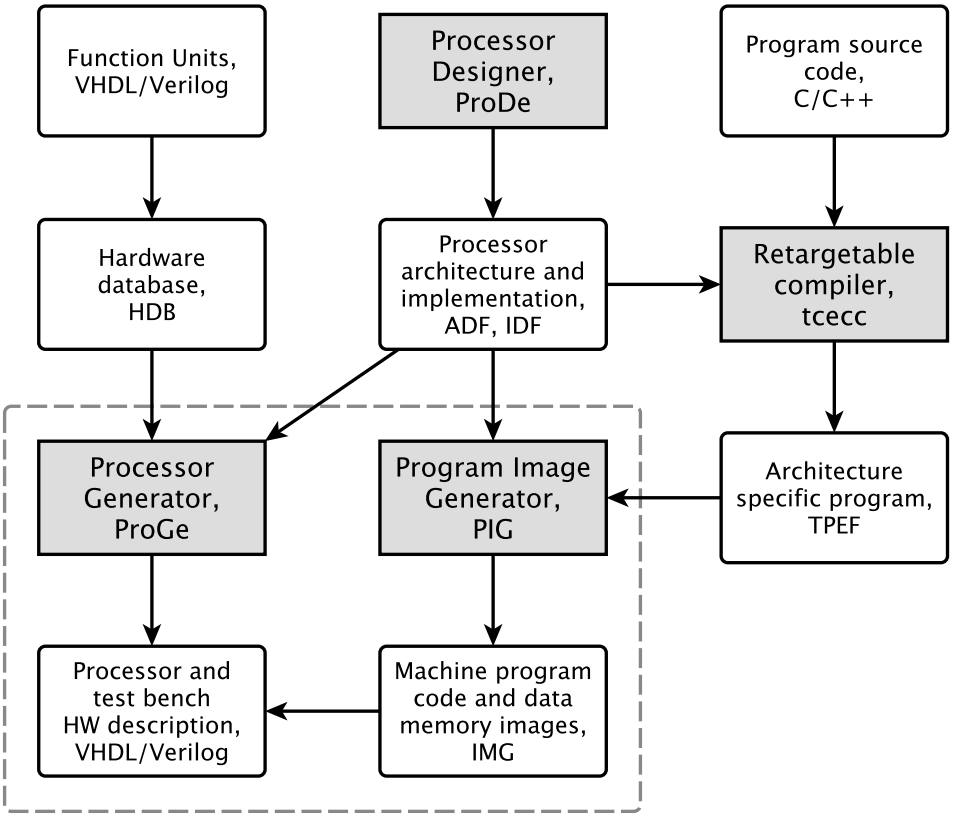

Figure 6.1: A part of the TCE toolset, containing the programs which have relevance to this work. Grayed boxes contain programs and white boxes contain the data formats associated to the programs. The programs and formats inside the dashed region undergo changes in this thesis' software integration.

The retargetable *Low Level Virtual Machine* (LLVM)-based [37,38] TCE compiler *tcecc* is used to compile programs written in C and C++ languages. The output from the compiler can be TCE-specific sequential bitcode or an architecture specific parallel TTA program in the *TTA Program Exchange Format* (TPEF).

Finally, the *Program Image Generator* (PIG) converts the TPEF object into instruction memory and data memory images, ready to be loaded into synthesizable hardware generated by ProGe. The format of the program image can be selected by giving PIG the desired input parameters. An important PIG parameter to note is the compression plugin, which can be set to use different compression algorithms on the program code, such as instruction template-based compression or dictionary compression. By default, no compression is used and the program image is written as *American Standard Code for Information Interchange* (ASCII) in the format of one memory word per line. Since the instructions are fixed length if no compression is used, one TTA instruction is written per line in the memory file.

6.2 Toolset Changes

The changes to take the new hardware into use were chiefly made to the programs ProGe and PIG, which are responsible for TCE's HDL generation and memory image generation. Both of these programs rely on the information found in an important, mostly under-the-hood component, the *Binary Encoding Map* (BEM). Below are the changes to the toolset, starting from the BEM.

6.2.1 Binary Encoding Map

Inside TCE, the processor's architecture definition is converted into a BEM object. The binary encoding map describes how to encode TTA instructions for a given target processor into bit patterns which make up the program image file generated later in PIG. The BEM also contains the encoding information for each instruction template and an alignment bit is automatically added to the instruction encoding if variable width templates exist. Before the generation of HDL or program and data images, the ProGe and PIG check from the BEM object whether variable length instruction templates have been defined. ProGe and PIG were modified to automatically generate HDL and memory files according to the new hardware designed in this thesis, if any templates of varying lengths have been defined.

6.2.2 Processor Generator

The processor generator is in charge of generating most of the HDL in the TTA processor, such as the TTA top level entity and function units, but more importantly, creating the fetch unit and the decoder. These both change significantly when variable length instructions are taken into use. ProGe checks whether variable length instruction templates have been defined in the BEM and automatically creates the new files required for variable length instruction support.

The hardware files are created through function calls from *ProcessorGenerator* class's *generateProcessor* function. The associated most important function calls for processor generation are displayed in Figure 6.2.

The *netlistGenerator.generate* function begins the generation of the top-level netlist model. It was not changed for the implementation. It gathers the port connections in the top-level entity for all the units that are instantiated at the top-most TTA processor level into the object model, such as the ports of the fetch, decode, decompressor, function units, register files and immediate units. The latter three are looked up from the IDF file.

Plugin.generate function is called next, which generates a large share of the required HDL for variable length instruction encoding. This plug-in is named *ICDecoderGenerator* and has the task of generating the interconnection network, the

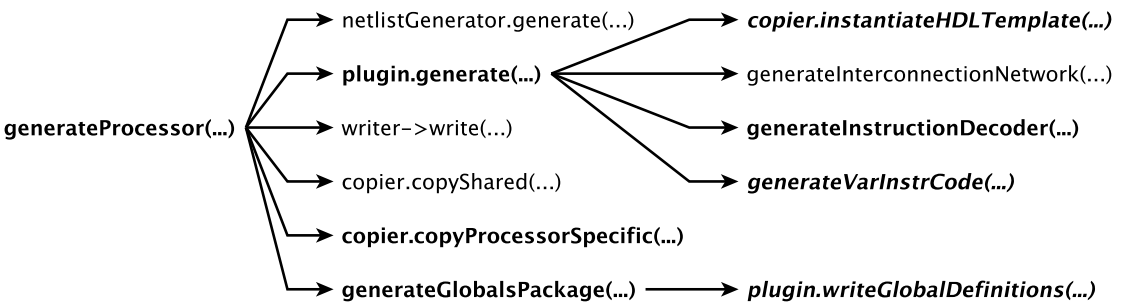

Figure 6.2: Function call diagram for processor generation, where modified functions are in bold and new functions are in italics and bold. The functions are displayed in the sequential execution order.

decoder and additional hardware such as a debugger based on parameters settable in the IDF. Whether to create a ring buffer fetch or a shift register fetch was parametrized and given as a task for the plug-in.

The plug-in calls the *copier.instantiateHDLTemplate* function which generates a RB or a SR fetch depending on a `VAR_LENGTH_FETCH_TYPE` parameter, which can be set in ProDe or written to an .IDF. When this parameter is `'sr'`, the SR fetch is generated. With any other input the RB is generated, if variable length instructions have been affirmed. The fetch unit's .vhdl-file is copied from a template file, where only a string for TTA processor's entity name is changed to an actual entity name.

The *generateInterconnectionNetwork* function generates the processor's interconnection network, which is tied directly to the decoder. Respectively, *generateInstructionDecoder* function generates the decoder and additional logic for the instruction template decompression. The template decompression is done by looping through all the instruction templates and within those looping through each move slot to see whether it is a NOP slot or a move slot. For NOP slots, the full uncompressed instruction's bits are taken from a NOP constant, which contains the correct NOP bits for each slot. For moves, the bits are taken from the actual instruction that arrives into the decoder. Care must be taken here, because the instruction's move slot's position is different if any slots prior to the current move slot are NOP slots, as shown back in Figure 5.8.

The *generateVarInstrCode* function is the last function called by the plug-in. This new function adds two new ports to the netlist's model, which are required in the final top-level netlist between the fetch and decompressor units.

The netlist writer's unmodified *write* function is called after the plug-in has added all the required extra ports and signals to the top-level entity model. The writer creates the actual top-level VHDL or verilog file depending on which language has been chosen in ProGe's arguments, being VHDL by default. The variable length instruction fetch and decompression have only been designed in VHDL.

The *copier.copyShared* function copies the HDL implementation files of the FUs, register files and immediate units into the processor's sub-folders. It did not require changes. However, the *copier.copyProcessorSpecific* function copies the fixed length fetch unit's, a decompression unit's and an opcode package's HDL files to their respective directories. This function was changed so that if the variable length fetch unit has already been copied by the *plugin.generate* function, the fetch copy operation is ignored inside this function call. The decompressor unit copied by this function is merely a dummy unit, which does no decompression and just lets the variable length instructions pass from the fetch unit to the decoder, where the actual instruction template decompression is done. The dummy decompression unit exists to be replaced for other decompression models, such as for dictionary compression.

Finally, the *generateGlobalsPackage* function creates the HDL-file for the global constants required by the HDL around the TTA processor. This consists of constants relating to the instruction memory width. Furthermore, a call is made to the *ICDecoderGenerator* plug-in's new *writeGlobalDefinitions* function, which is now in charge of the instruction width constant in the TTA processor. In the case of variable length instructions, some additional constants are written: widths of all instruction templates inside a table, the amount of templates, and the width of the instruction template field.

This concludes the generation of the processor's HDL files based on the architecture and implementation through the *generateProcessor* function. By default, a simple test bench is also copied over when the processor is generated through *ProDe*, or with the `-t` argument given to the *generateprocessor* program.

6.2.3 Program Image Generator

PIG is a tool in TCE which creates the data and program images from an architecture specific binary assembler program file (.tpef). The image files are loaded into simulation by the test bench, which *ProGe* generates alongside with the TTA processor itself. In addition to a few command line arguments which were added for *PIG*, a new compressor plug-in for instruction templates and a variable length ASCII program image writer was implemented. Just like *ProGe*, if *PIG* detects that variable length instruction templates have been defined in the BEM, the new compressor and image writer are used automatically. Two new command line argument values were added:

Program image format argument `-f` can now get the value `varascii`, which means to create the program image using the new variable length ASCII program image writer. If the argument is not given, `varascii` is used by default if variable length instruction templates have been defined. Otherwise the default is `ascii`.

Compressor plugin argument `-c` can now get the value `NOPTemplateCompressor.so`, which tells PIG to use the new compression plug-in in accordance to the NOP removal instruction templates defined in the ADF. Using this compressor necessitates using `-f varascii`. This compressor is used by default if variable length instruction templates have been defined. Otherwise the default compressor is used.

NOP Template Compressor Plug-In

NOP Template Compressor is a new instruction compressor plug-in, which generates a program image bit vector. The bit vector is constructed by calling a method, which turns the program's instructions one at a time into bit patterns according to the BEM. In addition to this basic bit vector generation task, which is done by the default compressor as well, the new plug-in has to take care of two new tasks: First, find out which instructions are jump and call targets so they can be aligned to start at the beginning of a memory address in the program memory. Second, apply the alignment bit to the instructions prior to the jump target and call target instructions. The fetch unit then knows to disregard the bits in the memory word in those instructions which have an align bit, as explained in Figure 3.1.

The jump targets are found from the so-called *instruction reference manager*, which records a reference for each instruction that is jumped to. While adding the instructions to the bit vector, a check is made at each instruction whether the next one has a reference. If true, the alignment bit is set to '1' in the current instruction.

For calls, there are a number of delay slot instructions separating the call instruction and the instruction which is returned to from the call's subprogram. Thus each instruction which is *pipeline delay slots* + 1 away from a call instruction is set to start at the beginning of a memory address. As the certain instructions for call targets are set to start at the beginning of a memory address in advance, a check can be made for them so that the instructions prior to call targets have their alignment bit set to '1'.

Variable Length ASCII Program Image Writer

The variable length ASCII program image writer outputs an image file from the bit vector provided by the NOP Template Compressor plug-in. This writer takes into account that the memory has a certain width and that variable length instructions must be a continuous stream in the memory, wrapping from one line to the next. In addition, the last memory line is padded with zeroes to match the memory width and two extra lines of zeroes are also added. The purpose of the extra lines is to prevent simulation faults from over-indexing, as the new fetch designs read two extra memory lines into their buffer at the end of the program.

7. VERIFICATION AND EVALUATION

This chapter first talks about the approaches taken to testing the correct functionality of the hardware fetch units designed. The second section presents an evaluation of the work done in this thesis with a single case study: A TTA processor is created and the compression ratio and resulting power savings are estimated in a CHStone test suite [5] using two different NOP removal template configurations. Then the power consumption of the ring buffer and shift register fetch designs for variable length instructions are estimated through synthesis and simulation on a 40 nm ASIC technology node. A comparison is made to see if the compression efficiency is sufficient to reduce the memory power consumption enough to justify using variable length instructions on a TTA processor.

7.1 Verification and Testing

The design process of complex hardware designs necessitates verification to ensure correctness. Additionally, the hardware description language written should conform to certain rules so that the design can be synthesized for both *Field-Programmable Gate Array* (FPGA) and ASIC technologies. Several simulation, synthesis and verification tools are required to ascertain design correctness and code compatibility.

In this work, the two main simulation tools used for verification were ModelSim [36] and GHDL [35]. Synopsys Design Compiler [39] was used to guarantee code compatibility with ASIC technologies and Altera Quartus II [40] to test synthesis for FPGA platforms. The two latter tools were also used to gauge the designs' power consumption, area and performance data throughout the design process. Especially Quartus was used to see whether certain design choices would improve or worsen the fetch unit's performance and area results.

7.1.1 Verification Process

The verification process was continuous and simultaneous with the design of the fetch units. As the designs were being written, their functionality was incrementally tested against the already existing fixed length instruction fetch unit in small test cases. Testing time line is presented in Table 7.1, which lists what test cases were used at each design phase of the fetch units. The test types found in the table are detailed in Chapter 7.1.2.

Design phase	Tests used
Basic functionality	Hand-written custom program image tests
Full functionality without jumps and calls	complex_multiply test and c2vhdl test using -O3 flag to remove jumps and calls, used as bus trace tests
Full functionality with jumps, no calls	complex_multiply test and c2vhdl test using -O0 to get longer test cases, used as true or false tests
Full functionality	complex_multiply test and c2vhdl test using all optimization parameters, Full CHStone test suite
Optimization and measurements	Full CHStone test suite

Table 7.1: Tests used during each design phase of the fetch units.

Every time a new piece of functionality was added, the design under testing could be subjected to a larger amount of tests with increased complexity. Furthermore, longer test cases more likely to encounter bugs could be created out of the same C-code by giving *tcecc* reduced optimization parameters, such as `-O1` and `-O0`. When design maturity was reached, in terms of functionality, the CHStone test suite's each test was run using the variable length fetch units. During the optimization phase and when increased functionality was decided to be added after design maturity, a python script called `hdl_tester.py` was written and integrated to TCE for quickly running a subset of the CHStone suite on the design with desired parameters. The instruction template-based compression was added at the very end of the design phase and its decompressor was tested separately using the CHStone test suite.

7.1.2 Test Types

Custom Program Image Tests

At the very beginning of the hardware design phase, the custom fetch unit was not advanced enough to be able to take in an entire program image that had been created with TCE from a C-program. Not only that, but there were no tools to create compressed and properly aligned program images automatically yet. The program image had to be written by hand instead, but was not a very difficult task if a very simple TTA processor just containing a single bus was created. These tests were used to see if the fetch unit could handle fetching and dispatching a few simple variable length instructions forward to the decoder, and were soon made obsolete when the TCE's *PIG* was improved to be able to create program image files in the proper format.

Bus Trace Tests

When most of the fetch units' basic functionality was in, simple bus trace tests could be used for verification. The *complex_multiply* and *c2vhdl* tests found from TCE were used for this purpose. These tests could be stripped from all jump and call instructions except for the end loop by compiling the C-test with *tcecc* using the `-O3` flag, as it simplified the resulting program.

In a bus trace comparison test, all the values from a single signal of the variable length instruction processor were written into a file. This file was then be compared to the signal of a processor using the original fixed length fetch unit. The signal chosen to be bus traced was the *Load-Store Unit's* (LSU) output port, which often communicates data and values that would be different if the execution of the new fetch unit differed at all from the original. Usually a difference in the signal meant that the program execution veered off to a wrong part in the program memory. However, the LSU also communicated jump and return call addresses, which would naturally be different on a variable length instruction processor instead of the fixed length kind due to program memory compression. Thus bus tracing the LSU's ports could no longer be used when jump functionality was added into the fetch unit.

True or False Output Tests

After bus tracing as a verification method was invalidated, all the tests were made to output a simple *True* (T) or *False* (F) character into a file instead if the C-program executed correctly. This was done using a *hdl_stdout*-unit, which could write characters to a text file during HDL simulation. All of the later tests in the design cycle were of this kind: *complex_multiply*, *c2vhdl* and CHStone suite's tests.

CHStone Benchmark Suite

CHStone [5] is a C-based high-level synthesis benchmark suite with programs from various application domains. Each of them contains C-code which is translated into a program image using *tcecc* and *PIG*, and outputs a T or F value at the end of the test. A subset of the test suite has been ported for use in TCE. The three shortest tests, which were still in thousands of clock cycles, were integrated into TCE's long system test regression for testing the correctness of the fetch units. These tests *gsm*, *mips* and *motion*, were good verification tests for their program nature: they all consist of many jumps and calls, which stress the complex jump timing control logic of the fetch units. Particularly the *mips* test is essentially a large switch-case program which turns into mostly serial code with a lot of jumps. The remaining longer tests, which were also used for verification and benchmarking but were not added to regression due to their length, are *adpcm*, *aes*, *blowfish*, *jpeg* and *sha*.

7.2 Evaluation of Results

A TTA processor with an I_{max} of 256 bits was created for a subset of the CH-Stone test suite to measure the compression efficiency of two different instruction template compression configurations: four and eight templates. In these configurations, two and six templates were used for NOP removal, respectively. The power consumption of the program memory was estimated with CACTI 5.3 [6] pre- and post-compression. The power consumption of the two fetch designs were estimated with three different quanta using synthetic tests to scope out the worst case power dissipation. Additionally, the used chip area of the designs is provided.

7.2.1 Compression Efficiency

In these results, the compression efficiency is reported as space saved by compression:

$$Space\ saving\ (\%) = 1 - \frac{Compressed\ size}{Uncompressed\ size} \times 100 \tag{7.1}$$

CHstone test suite was used for measuring the compression efficiency of the instruction template-based compression. A TTA machine with an I_{max} of 256 and a q of 32 was customized for the benchmarks. The processor was created by starting with a 6-issue VLIW equivalent processor architecture and reducing it by combining rarely used buses until a 256-bit instruction length was reached. The benchmark programs' uncompressed sizes were in the range of 14–50 KB, with the exception of the *jpeg* test which was approximately 376 KB.

Greedy workload-based template selection was used: A program was scheduled on a TTA machine without any templates first. Then all possible templates were iterated through with a given number of NOP slots and the one which could represent the most instructions in the program was selected. Subsequent templates were selected based on how much they improved the number of covered NOP moves. The narrowest templates are optimized first, since they give a better compression ratio.

For the machine with 2 NOP templates, a 64-bit and a 128-bit template optimized for the *adpcm* benchmark were selected using the greedy workload-based selection process. For the machine with 6 NOP templates, a 64-bit and 128-bit templates optimized for the *gsm* program were added, since it had the weakest compression ratio, as well as two 32-bit templates optimized for *adpcm*. The 32-bit templates were placed on buses controlling *load-store unit* and *control unit* trigger ports, which are likely to be used in serial code.

Resulting memory space savings are shown in Figure 7.1. The 2-template TTA reached an average program size reduction of 37% and a maximum of 46%, and the 6-template TTA improved to an average of 44% and a maximum of 51%.

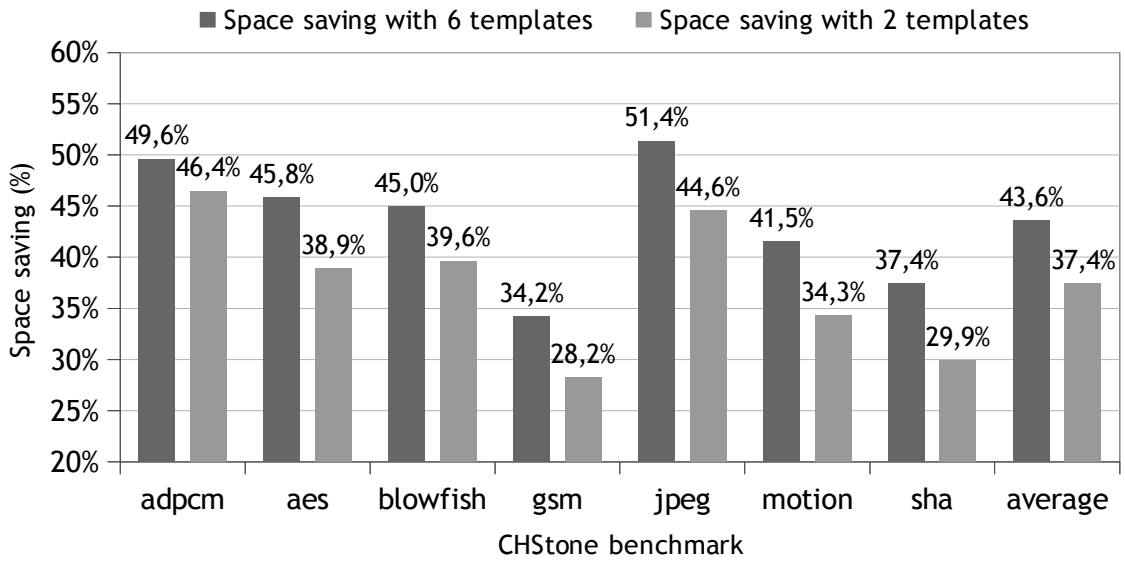

Figure 7.1: The amount of memory space saved in the CHStone benchmarks.

7.2.2 Program Memory Power Consumption

The power consumption of the program memory was estimated with CACTI before and after instruction template-based compression. LSTP was chosen as the SRAM transistor type, interconnect projection type was set to conservative and wire outside of mat as semi-global. Technology node used was 40 nm and temperature was set to 300K for all measurements. The number of bits read out was matched with the memory width, that is, the full instruction length of 256 bits. One read/write port was used. For the estimation, SRAM size was set exactly to the size of the program, which is unrealistic as SRAM is not manufactured in arbitrary sizes, but gives an estimate of power savings achieved by the instruction template compression. Finally, the total *dynamic read power per read port* (P_{dyn}) is calculated with

$$P_{dyn} = \frac{E_{dyn}}{t} = E_{dyn} f_{clk} \tag{7.2}$$

where E_{dyn} is the *dynamic energy per read port* estimated by CACTI and f_{clk} is the *clock frequency* of 600 MHz for the SRAM, which is also the target frequency used in the synthesis of the fetch units later in Figure 7.4. Since LSTP SRAM cells were used in the measurements, the portion of leakage power was much less than 0.1% of the total power consumed and could be left out of consideration. The overhead of the instruction template bits and padding bits required by the proposed TTA's variable length instruction format are taken into account in the results, while their effect is minimal ($< 1\%$).

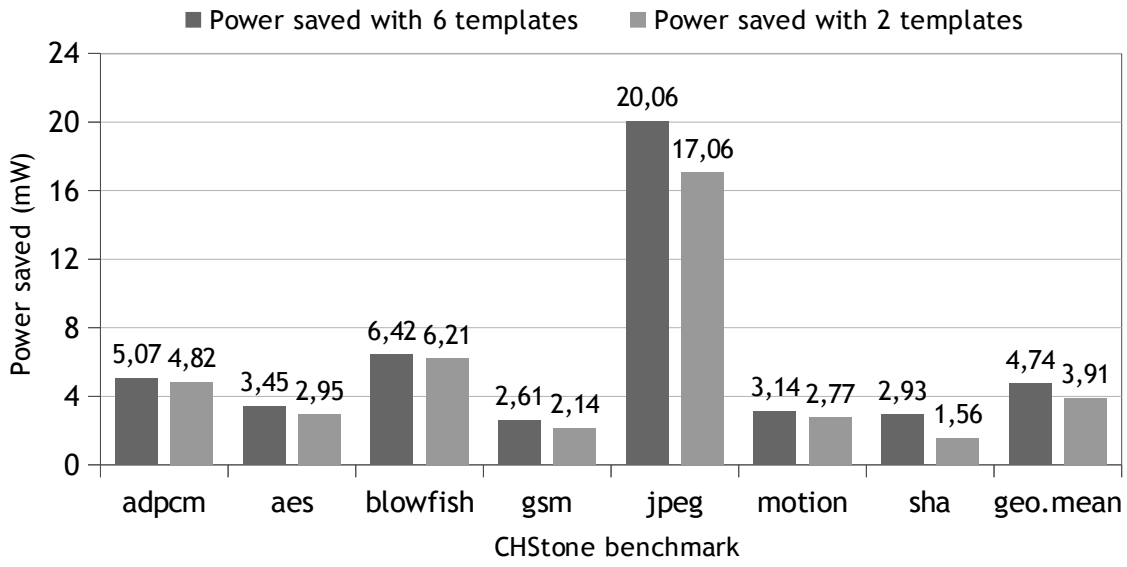

Figure 7.2: Power saved with instruction template compression, using 2 and 6 instruction templates.

The power savings per CHStone benchmark are presented in Figure 7.2. The difference between 2 and 6 instruction templates used for the NOP removal is also visible in the power results: 6 templates covered more of the NOP moves, allowing better compression ratio and smaller SRAM memory size. In order to compensate for the *jpeg* test results, where the benchmark contains a significantly larger instruction count, a geometric mean of the power saved in all the tests is presented: 4,74 mW with 6 instruction templates and 3,91 mW with just 2 instruction templates. The power saved was not linear with the amount of bytes reduced from the program code, because the size of the program memory affects the consumption, especially when power of two values are crossed. Despite approximately 21 KB was saved in the *aes* test with six templates, only 3,45 mW less power was consumed, while 6.42 mW of power was saved in the *blowfish* test with 13 KB memory reduction. As examples, the program code for *aes* could now be fitted on a 32 KB memory instead of 64 KB after compression, and *blowfish* on 16 KB instead of 32 KB.

Since SRAM memory is not manufactured in arbitrary sizes and the power of two sizes have such importance, the power saved when switching to a half smaller memory size was estimated with CACTI with the same parameters as for instruction compression. These results are presented in Figure 7.3. The chart shows that a considerable saving is seen each time when a reduction is possible, until 16 KB. This highlights that a good amount of power can be saved even if the program image does not compress significantly, but if it compresses sufficiently to fit on a smaller memory module.

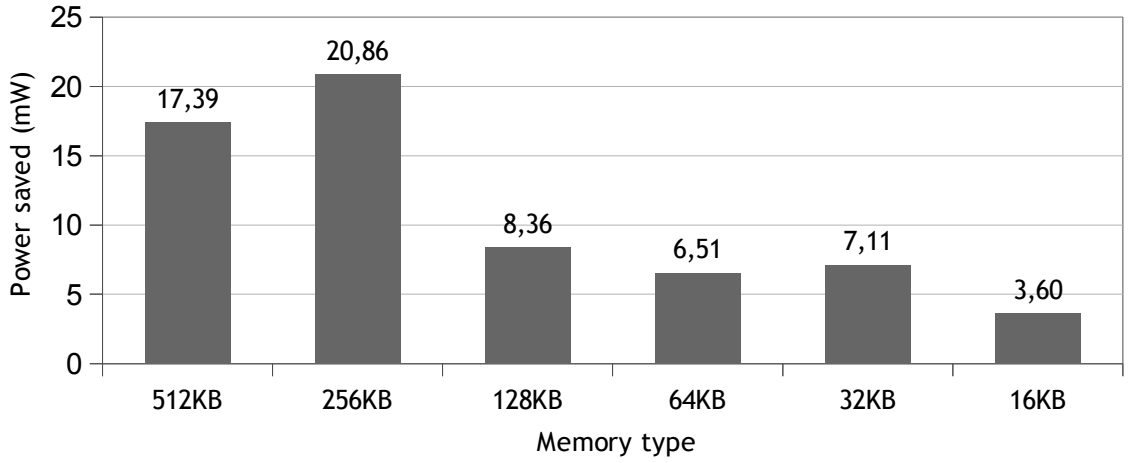

Figure 7.3: Memory power saved when LSTP SRAM memory size is reduced by half.

7.2.3 Fetch Unit Power Consumption

The original fixed length and the two variable length fetch designs, ring buffer and shift register fetch, were synthesized on a 40 nm standard cell technology, using quanta of 2, 32, and 128 bits. The target clock speed was set to 600 MHz, as the LSTP SRAM cells functioned well until that range according to CACTI. Each variable length design variant was subjected to three synthetic test cases, which explored the units' worst case power consumption. The three test cases consisted of a varying degree of I_{max} = 256-bit and q-length instructions: Either all q-length, all I_{max}-length or alternating I_{max}- and q-length instructions. These synthetic tests stressed the fetch units' internal shifter and multiplexer logic used for manipulating the variable length instructions within the buffer.

The test result with the highest power consumption for each design variant is displayed in Figure 7.4. In most cases, the worst power consumption was seen when the fetch units had to repeatedly fetch and handle q-length instructions, as their internal multiplexer and shifter structures had to operate on bits. The best results are seen with a q of 128 bits, which is half of I_{max}. The RB design requires 3,50 mW of extra power at minimum at $q = 128$. The smaller q-values follow closely, while the power consumption grows rapidly on the SR design if the smallest instruction size is reduced. This tells that the RB design is better for small instructions on a large processor, which allows for a better compression ratio.

At lowest, the SR variable length fetch unit requires 3,84 mW of extra power to operate when the q of 128 bits is used. However, a much better compression ratio is seen with a q of 32 bits, which is the quantum used in the instruction template compression results earlier in Figure 7.1. The SR approach for a q of 32 consumes more power than would be saved with 6 instruction templates on average, unless a reduction from a 128 KB memory or larger to a smaller category can be made.

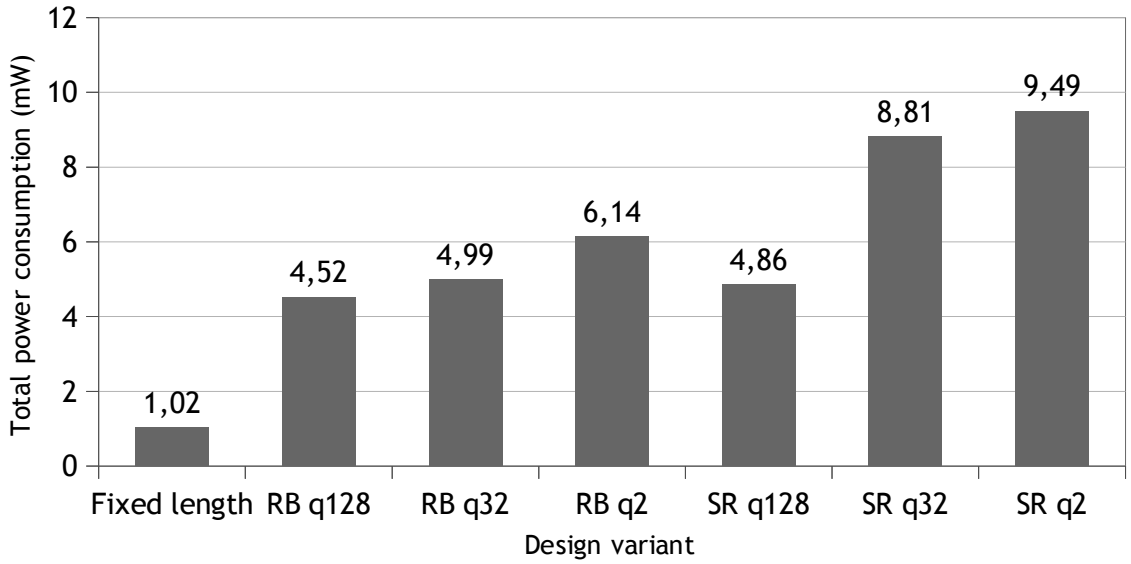

Figure 7.4: Fetch units' total power consumption with quanta (q) of 2, 32 and 128 bits, showing worst case test results for each design variant.

The RB fetch unit is much more efficient, reaching the break-even of average power savings when just two instruction templates are used for compression.

To put these power values into perspective, each of the synthesized variable length fetch units consumed 65,5% – 79,5% of the entire TTA processor's total power at f_{clk} = 600 MHz. In the worst case synthetic test for each fetch design variant, the TTA processors consumed power between 7,0 mW and 11,9 mW. In a fixed length instruction processor, the fetch unit consumed only approximately 31,5% of the total power of 3,3 mW. The power consumption impact from adding a variable length fetch unit is quite severe, and only justifiable with good power savings from sufficient compression.

As long as a SRAM memory power saving of approximately 3,50 mW or more is reached with compression, the variable length RB fetch unit's usage is favorable. These results do not include the overhead from the instruction template decompression which is integrated in the decoder unit, which consumes additional dynamic power to re-assemble the decompressed instructions. This can be projected to be a fairly efficient operation, as it is a multiplexer network which simplifies by choosing a reasonably large q and using few instruction templates.

7.2.4 Chip Area

The area of each of the fetch designs was collected from the 40 nm standard cell synthesis results and is presented in Figure 7.5 in kilogates. A similar trend is seen in the area as in the power consumption: the SR designs with a small q grow rapidly, while the ring buffer stays more compact even when q is increased. Worth

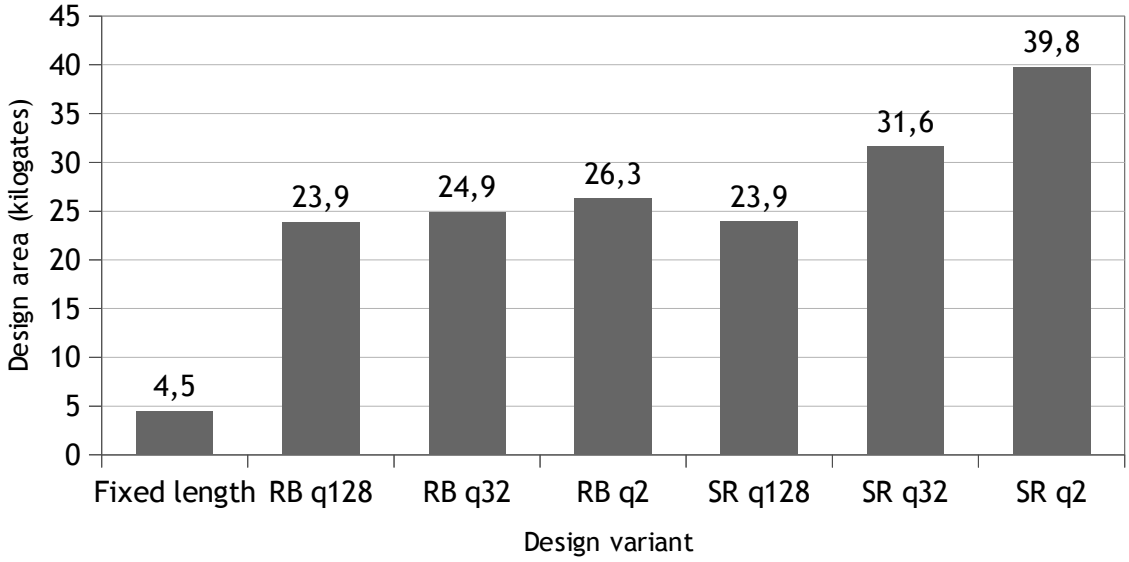

Figure 7.5: Fetch units' area in kilogates with quanta (q) of 2, 32, and 128 bits, using target clock speed of 600 MHz.

noting is that SR design's area exploded when the maximum instruction length of a *power of two value* − 1 was used, while the ring buffer's area followed a near linear trend with maximum instruction size increase. It is interesting to note that at their simplest form at q of 128, the RB and SR are of similar size. This implies that the extra logic the RB requires to function roughly equals the extra logic required by the SR design's buffer, which is one instruction longer. Finally, with the least logic generated with a q of 128 bits, both of the new designs are 431% larger than the original fetch design, which only handles fixed length instructions.

7.2.5 Summary

The compression achieved 44% program size reduction on average with 6 NOP removal templates and 37% reduction with 2 templates. The fetch designs consume an extra 3.50 mW of power at minimum on a TTA processor with 256-bit maximum instruction length, when ring buffer fetch is used. The overall performance in terms of clock speed remains the same as with a TTA processor using fixed length instructions, up until approximately 1 GHz, where the ring buffer fetch starts having setup time violations during ASIC synthesis with a very small q.

Despite the savings from the template compression do not always directly surpass the extra power consumed by the fetch unit in this benchmark, the target program could often be fitted on a half smaller memory module after compression. In these measurements with the 256-bit TTA, for SRAM memory sizes between 32–512 KB and beyond, the power consumption reduction through compression is sufficient to benefit from the variable length instruction architecture.

To summarize on the fetch units' differences according to the results, the RB and SR units are approximately the same size and consume nearly the same amount of power when I_{max} is the same and $I_{max} = 2q$. When q is reduced, the instructions become smaller and can be aligned in more numerous ways inside the fetch units' buffer. Because the SR's buffer is wider by one I_{max}, its area and power consumption grows faster than RB's. How significant the growth of the power usage was, is surprising. The one extra buffer slot allows SR to have simpler internal control logic, which shortens its critical path and consequently leads to better performance values in terms of clock frequency.

Since the fetch units' extra power consumption is so close to the power amount saved, it cannot be said yet that the compression will always be beneficial. Moreover, while estimated to be small, the impact of the decompression procedure on the power consumption remains unknown until benchmarked.

8. CONCLUSIONS

In this work, two variable length fetch units and an instruction template decompressor were designed and implemented from scratch in VHDL. The designs were integrated to the TCE toolset for use in future processor prototyping. The fetch units and the decompressor can be used for compressing away excess NOP operations on the instructions in TTA architectures by using variable length instruction encoding and instruction template-based compression. The compression approach aims to reduce the SRAM memory size required for the program code, lowering the total power consumption of the processor despite requiring additional hardware for the fetch and decompression stages. The designs took both ASIC and FPGA compatibility into account, but were mainly developed for use in actual low-power embedded integrated circuits using modern deep submicron technologies.

The first fetch design is the ring buffer, which strives for minimal power consumption through minimum possible instruction buffer size. The buffer can contain two instructions of the maximum size available in the processor architecture. Due to this restriction, the control logic became complex and the design requires an additional lock cycle after every jump or call to reset its internal counters and control signals. Furthermore, the design has somewhat limited clock speed due to a long critical path, but is fast enough to function up to 900 MHz with quanta above 2 bits in a 256-bit-instruction TTA machine synthesized on a 40 nm ASIC technology. Its power consumption turned out to be minimal, requiring only 4,5 mW of power compared to the original fixed length fetch unit's 1,0 mW when the quantum is half of the maximum instruction size. Even when the q is reduced, the power consumption does not ramp up quickly, as seen from Figure 7.4. This power requirement is often less than the power saved through instruction compression, making it a viable design to use in fetching instructions with varying lengths.

The second fetch design, the shift register buffer, addresses the complexity issues of the RB design. It does not require using one extra lock cycle after jumps and calls due to its buffer, which is one maximum instruction longer than in the former design. The simplified control logic comes at the price of overall larger hardware footprint due to the increased buffer size. The end result is that with a large quantum which is half of the maximum instruction size, the design rivals the former ring buffer alternative with a power consumption of 4,9 mW, but the power consumption

increases rapidly when the q is lowered, as seen from Figure 7.4. The SR design does offer higher performance, but was only synthesized up until 1 GHz in the tests. This fetch design could still be useful in a processor which is not as strict on the low power requirement, aims for higher clock speeds and uses high performance SRAM to support the fetch unit's speed. The exact areas of justified use remain unproven as not enough power consumption benchmarks could be performed, owing to time constraints and lack of conveniently available power measurement tools at the university's department, where the work was carried out.

The decompression scheme which was integrated directly in the TTA processor's decoder was left unbenchmarked for the same reasons as the thorough exploration of the SR design's use areas: unavailability. It was estimated to be a power-efficient procedure if the minimization of the amount of templates, the maximum instruction size and the template sizes are taken into account in the architectural design phase.

To summarize, the ring buffer fetch unit performs very power-efficiently around the 600 MHz clock speed for handling variable length instructions, making it justifiable to use together with program code compression according to this thesis' benchmarks. The shift register fetch design leaves a desire for enhancements until it can be used for very low power applications. Many variables, which can be adjusted during the processor's architectural design phase, are at play for affecting the fetch units' final power consumption values. The memory power consumption varies greatly based on the program size and SRAM parameters, and the synthesized TTA processor's power consumption depends on many aspects, such as switching activity, transistor types, design size and target performance.

Ultimately, the question whether variable length instruction encoding is useful on TTA processors for NOP removal, receives a partial positive answer. But further benchmarking, which is outside of this thesis' scope, needs to be done to ascertain useful fields of application. Thorough case-by-case evaluation is required until more experience is gained on the power cost owing to variable length instruction encoding's hardware requirements.

As often with scientific work, several new ideas for improvement were encountered through epiphany during the design and implementation process. Specifically the hardware design of the fetch units has room for improvement. Some small optimizations could be probably be done if the entire fetch code was rewritten once, especially for the ring buffer design, the HDL structure of which isn't quite tidy or well arranged. In addition to that, a few larger ideas came to mind as well.

The fetch unit could be integrated with a small *level 0 or 1 cache*, which would contain the few last memory words for power-efficient loop buffering. It could eliminate extra dynamic power needed to read the same instructions over and over again from the program memory. But it would result in additional hardware, which needs

to be benchmarked for power consumption versus the power saved from memory reads, much like the variable length fetch units in this work. The size of the cache would have to be customizable for the architecture and program, which affect the size of the data crunching loops.

More detailed power benchmarks is another necessary task to be done. Only a 256-bit-instruction TTA processor with variable length fetch units was synthesized and simulated on a 40 nm standard cell library for power measurements. The amount of power saved from SRAM reductions drops significantly when the program memory size becomes small. It is interesting to see whether the variable length approach is also viable in small TTA processors used in ultra low power applications, such as sensor node systems. Furthermore, the instruction template decompression's power consumption is yet unmeasured due to time constraints and unavailability of tools at the department, and the fetch units were only tested for power consumption with synthetic test cases instead of real world applications.

Ultimately, the next big advancement in the future is using the variable length instructions for other encoding tasks as well, not only for NOP removal. Some operations require less operands or payload than others, such as short immediates, which can be taken advantage of in order to reduce instruction sizes even further. The fetch units can already handle instructions of any size, but a considerable challenge in implementing a more advanced variable length instruction format is deciding on the efficient encoding of the ISA and designing the decoding architecture for multiple instruction formats.

REFERENCES

[1] L. Sheng, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of Annual International Symposium on Microarchitecture*, New York, NY, Dec. 2009, pp. 469–480.

[2] B. S. Deepaksubramanyan and A. Nuñez, "Analysis of subthreshold leakage reduction in CMOS digital circuits," in *Proceedings of Midwest Symp. Circ. Syst.*, Montreal, QC, Aug. 2007, pp. 1400–1404.

[3] H. Pilo, C. A. Adams, I. Arsovski, R. M. Houle, S. M. Lamphier, M. M. Lee, F. M. Pavlik, S. N. Sambatur, A. Seferagic, R. Wu, and M. I. Younus, "A 64Mb SRAM in 22nm SOI technology featuring fine-granularity power gating and low-energy power-supply-partition techniques for 37% leakage reduction," in *Proceedings of IEEE International Solid-State Circuits Conference Digest Tech. Papers*, San Francisco, CA, Feb. 2013, pp. 322–323.

[4] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proceedings of SPIE Multimedia on Mobile Devices*, Jan. 2007, pp. 65 070X–1 – 65 070X–11.

[5] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, Oct. 2009.

[6] Hewlett-Packard Development Company. CACTI: An integrated cache and memory access time, cycle time, area, leakage and dynamic power model. [Accessed on 28th of Apr. 2014]. [Online]. Available: http://www.hpl.hp.com/research/cacti/

[7] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 51–62, Jun. 2008.

[8] J. Helkala, T. Viitanen, H. Kultala, P. Jääskeläinen, J. Takala, T. Zetterman, and H. Berg, "Variable Length Instruction Compression on Transport Triggered Architectures," Apr. 2014, accepted to International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS) XIV.

[9] J. A. Fisher and B. R. Rau, "Instruction-level parallel processing," in *Instruction-Level Parallel Processors*, H. Torng and S. Vassiliadis, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1995, pp. 41–49.

[10] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, Feb. 2014.

[11] H. Torng and S. Vassiliadis, *Instruction-Level Parallel Processors*. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[12] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, 5th edition*. Waltham, MA: Elsevier, Inc., 2012.

[13] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[14] H. Ando, M. Nakaya, C. Nakanishi, and T. Hara, "Performance comparison of ILP machines with cycle time evaluation," in *Proceedings of Annual International Symposium on Computer Architecture*, Philadelphia, PA, May 1996, pp. 213–224.

[15] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *ISCA '83: Proceedings of 10th International Symp. on Computer Architecture*, Los Alamitos, CA, Jun. 1983, pp. 140–150.

[16] *ADSP-TS201S TigerSHARC embedded processor data sheet (Rev C, 12/2006)*, PDF, Analog Devices, Inc., 2006.

[17] *Qualcomm Hexagon DSP: An architecture optimized for mobile multimedia and communications*, PDF, Qualcomm Technologies, Inc., 2013, [Accessed on 28th of Apr. 2014]. [Online]. Available: http://pages.cs.wisc.edu/~danav/pubs/qcom/hexagon_hotchips2013.pdf

[18] *Efficeon processor product brief*, PDF, Transmeta Corporation, 2004, [Accessed on 28th of Apr. 2014]. [Online]. Available: http://datasheets.chipdb.org/Transmeta/TM8600/efficeon_tm8600_prod_brief.pdf

[19] M. S. Schlansker and B. R. Rau, "EPIC: An architecture for instruction-level parallel processors," Hewlett-Packard, Tech. Rep., Feb. 2000.

[20] H. Corporaal, "Transport triggered architectures, design and evaluation," Ph.D. dissertation, Delft Univ. Tech., Netherlands, 1995.

[21] G. Lipovski, "The architecture of a simple, effective control processor," in *2nd Euromicro Symposium on Microprocessing and Microprogramming*, Oct. 1976, pp. 7–19.

[22] H. Corporaal, "Design of transport triggered architectures," in *Proceedings of Great Lakes Symposium on Design Automation of High Performance VLSI Systems*, Mar. 1994, pp. 130–135.

[23] J. Hoogerbrugge and H. Corporaal, "Register file port requirements of transport triggered architectures," in *Proceedings of Annual International Symposium on Microarchitecture*, San Jose, CA, Nov.-Dec. 1994, pp. 191–195.

[24] O. Esko, P. Jääskeläinen, P. Huerta, C. de La Lama, J.Takala, and J. Martinez, "Customized exposed datapath soft-core design flow with compiler support," in *Proceedings of International Conference Field Programmable Logic and Applications*, Milan, Italy, Aug. 2010, pp. 217–222.

[25] T. Patyk, D. Guevorkian, T. Pitkänen, P. Jääskeläinen, and J. Takala, "Low-power application-specific FFT processor for LTE applications," in *Proceedings of International Conference on Embedded Computer Systems*, Jul. 2013, pp. 28–32.

[26] J. Heikkinen, "Program compression in long instruction word application-specific instruction-set processors," Ph.D. dissertation, Tampere Univ. Tech., Finland, Nov. 2007.

[27] C. Auth, "45nm high-k + metal gate strain-enhanced CMOS transistors," in *Proceedings of IEEE Custom Integrated Circuits Conference*, San Jose, CA, Sept. 2008, pp. 379–386.

[28] S. Aditya, S. A. Mahlke, and B. R. Rau, "Code size minimization and retargetable assembly for custom epic and vliw instruction formats," *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 4, pp. 752–773, Oct. 2000.

[29] S. Segars, K. Clarke, and L. Goudge, "Embedded control problems, thumb, and the ARM7TDMI," *IEEE Micro*, vol. 15, no. 5, pp. 22–30, Oct. 1995.

[30] V. Weaver and S. McKee, "Code density concerns for new architectures," in *IEEE International Conference on Computer Design 2009*, Oct. 2009, pp. 459–464.

[31] M. Weiss and G. Fettweis, "Dynamic codewidth reduction for VLIW instruction set architectures in digital signal processors," in *Proceedings of 3rd International Workshop Image and Signal Processing on the Theme of Advances in Computational Intelligence*, Manchester, UK, Jan. 1996, pp. 517–520.

[32] H. Pan and K. Asanović, "Heads and tails: A variable-length instruction format supporting parallel fetch and decode," in *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Atlanta, GA, Nov. 2001, pp. 168–175.

[33] G. Kane, *MIPS RISC Architecture*. Prentice Hall, 1992.

[34] Tampere University of Technology. TCE: TTA-Based Codesign Environment. [Accessed on 28th of Apr. 2014]. [Online]. Available: http://tce.cs.tut.fi

[35] T. Gingold. GHDL: Where VHDL meets gcc. [Accessed on 28th of Apr. 2014]. [Online]. Available: http://ghdl.free.fr

[36] Mentor Graphics. ModelSim - Leading Simulation and Debugging. [Accessed on 28th of Apr. 2014]. [Online]. Available: http://www.model.com

[37] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation Optimization*, Mar. 2004, pp. 75–87.

[38] The LLVM Team. The LLVM Compiler Infrastructure: LLVM Overview. [Accessed on 28th of Apr. 2014]. [Online]. Available: http://llvm.org

[39] Synopsys, Inc. Design Compiler 2010: Doubles Productivity of Synthesis and Place and Route. [Accessed on 28th of Apr. 2014]. [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/

[40] Altera Corporation. Quartus II Subscription Edition Software: #1 Design Software in Performance and Productivity. [Accessed on 28th of Apr. 2014]. [Online]. Available: http://www.altera.com/products/software/quartus-ii/subscription-edition/qts-se-index.html